Escuela Politécnica Superior

Desarrollo de mecanismos de autentificación avanzados

Máster CiberSeg

Trabajo Fin de Máster

Autor:

Luis Pérez Sevilla

Tutor/es:

Rafael Ignacio Álvarez Sánchez

Junio 2019

Universitat d'Alacant
Universidad de Alicante

Índice

Índice de figuras

1. Introducción

Estamos en una sociedad de continuo desarrollo, la cual cada vez está más conectada gracias el rápido avance de la tecnología. Avanzamos hacia un futuro lleno de servicios en cada rincón y aspecto de nuestras vidas, accesibles al momento y desde cualquier lugar.

La tecnología debe avanzar hacia un futuro en el cual se protejan los derechos y privacidad de los usuarios en todos los ámbitos de la tecnología, ya sea Internet, aplicaciones móviles, servicios…

¿Cómo proteger la privacidad de los usuarios y la seguridad de los servicios al mismo tiempo? ¿Cómo identificar a los usuarios si poner en riesgo su privacidad?

En este punto, entra en juego la administración de identidades (en inglés, *Identity Management*). La administración de identidades es lo que se denomina un sistema integrado de políticas y procesos, mediante los cuales se pretende controlar el acceso a los sistemas de información [1].

Este concepto nace de la necesidad de proteger la información personal, bases de datos, aplicaciones y servicios; dentro del cual, podemos encontrar diferentes tipos de categorías con soluciones interconectadas para poder administrar la autenticación de usuarios, permisos, restricciones de acceso, perfiles de cuentas, contraseñas u atributos necesarios la administración de un usuario [1].

El problema que plantea este tipo de administración de identidades de manera digital es el hecho de que encontramos un único punto de fallo dentro de una organización, ya que normalmente todas las identidades son almacenadas y centralizadas en un servidor, además de que las identidades de los usuarios deben de ser privadas y seguras. Por todas estas razones, y con el avance y aparición de nuevas tecnologías, aparece una solución a este problema, *blockchain* como administrador de identidades.

1.1 Blockchain como administración de identidades

La tecnología *blockchain* permite que todo el mundo en una *red blockchain* posea el mismo nivel de fiabilidad sobre si las credenciales son válidas y quién en la red asegura la veracidad de ese dato, todo ello sin tener que revelar ningún tipo de información a ningún usuario que desee verificar dicha información.

1.1.1 Privacidad y seguridad en la administración de identidades

Gracias a como está construida la tecnología de *blockchain*, no hace falta que los datos provenientes de esta red sean validados. Ya que, debido a la propia naturaleza de la tecnología, dichos datos no pueden ser alterados una vez son escritos en la cadena de bloques.

Una de las características de la tecnología de *cadena de bloques* o *blockchain*, es la inmutabilidad, es decir, la información contenida dentro de ella no se puede modificar ni eliminar, por lo que, al no ser alterables, no se deberán de validar dichos datos o información.

Aunque los datos no se puedan modificar debido a las características de la tecnología *blockchain*, sí que deberemos de validar quién ha emitido dicho dato o información. Dicho de otra manera, en lugar de validar la fiabilidad del dato, habrá que validar la procedencia del dato, para determinar si se confía en quién lo ha emitido a la cadena de bloques o no. Por lo tanto, la validación se basa en la relación de confianza con el emisor de los datos.

1.1.2 Propietarios, emisores y validadores

Cuando se habla sobre *blockchain* como un administrador de identidades, se identifican siempre tres tipos de actores: los propietarios, los emisores y los validadores.

En este escenario, al primero que encontramos, será al emisor del dato o más bien al emisor de la identidad; o del dato identificativo. Es el encargado de validar la identidad del usuario y de crear la identificación necesaria en la *blockchain*. Nunca se almacenará

información personal dentro de la cadena de bloques, ya que todo el mundo posee una copia de los datos.

El propietario de esa información generada será el usuario que desea ser autenticado y validado. El usuario en este caso almacena esa información en su *cartera* o *monedero* de direcciones *blockchain* (la cual se utiliza para almacenar pares de claves, se abordará este tema más adelante en el documento). Dentro del ámbito de la administración de identidades, se refieren a esta *cartera* como *cartera de identidad*. Dentro de esta cartera se almacenará la información que prueba la identificación del usuario.

Y, por último, encontramos a los validadores, los cuales serán los actores que querrán validar a un usuario. Son llamados validadores porque serán los encargados de validar a los emisores del dato. La fiabilidad de los datos, en este caso de la identidad del usuario, dependerá de la reputación e integridad del emisor.

En este caso, el propietario se autentica contra el emisor de la identidad, dicho emisor verifica, autentica y valida a dicho usuario y genera la información necesaria para la identificación del usuario, almacenándose dicha información en la *cadena de bloques* o *blockchain* y la referencia a esa información siendo almacenada por el usuario. Cuando el usuario desea autenticarse o acreditar la identificación a un tercero, este le proporciona la referencia a la información generada por el primer emisor en la *blockchain*. El verificador, no validará la información, si no que verificará quién ha sido el emisor de dichos datos y comprobará si confía en la reputación de dicho emisor.

Dada la naturaleza de la tecnología, y como hemos mencionado en los párrafos anteriores, no se debe almacenar información personal, ni privada, puesto que todo el mundo tendrá acceso a dicha información; y en tal caso, estaríamos poniendo en peligro la seguridad y privacidad de los datos de usuario.

1.2 Administrador de identidades con blockchain como concepto

El concepto que acabamos de ver viene a cubrir o solucionar los problemas derivados de la identidad digital. Con esta solución se pretende perseguir el concepto de identidad única y global para los usuarios, dentro del ámbito de Internet y servicios. Todo

esto está ligado a la idea de *Identidad Propia y Soberana* (más conocida por su nombre en inglés *Self Sovereign Identity*). Esta idea propone que sean los usuarios y organizaciones los que realmente posean el control sobre su identidad digital. La idea es poder decidir qué datos se comparten y con quién se comparten dichos datos.

Esta idea ha sido la precursora de tecnologías como *Sovrin* o *uPort*. Estas buscan mediante este concepto de administrador de identidades con *blockchain* crear mecanismos de identidad digital autogestionada y descentralizada. De esta manera, persiguen un concepto muy interesante, pero que plantea muchos problemas que hay que superar antes de que se implante en nuestra sociedad actual.

Por todo lo que hemos visto hasta hora en este apartado, el siguiente paso para avanzar en esta dirección, es aplicar el concepto de administración de identidades con *blockchain* a las tecnologías actuales de autentificación de usuarios, de esta manera lograremos avanzar en la dirección correcta para lograr una identidad digital *propia* y *soberana*.

2. Justificación y Objetivos

Como se ha podido observar, existe un nuevo camino para conseguir administrar las identidades de los usuarios, gracias a la reciente tecnología de la *cadena de bloques* o *blockchain*.

Con este proyecto se pretende conseguir un servicio de autentificación *Single Sign-On* de usuarios de manera descentralizada, gracias a la administración de identidades basada en la tecnología *blockchain*.

Este desarrollo partirá del proyecto de final de grado llamado "*Desarrollo de un servicio de autentificación de factor múltiple*" [2]. Este proyecto cuenta con un sistema de autentificación de usuarios para múltiples servicios, con segundo factor de autentificación biométrico. Con ese proyecto, se desarrolló un sistema, en el cual se garantizaba la privacidad de los datos del usuario mediante una buena gestión de estos en el sistema, proporcionando autentificación segura, rápida y personalizable para cualquier tipo de servicio o tecnología, con el fin de cumplir los requisitos de seguridad de cada servicio. En los siguientes apartados se describirán los diferentes aspectos de diseño, desarrollo e implantación que conforman este nuevo proyecto.

Todo el sistema de autenticación de usuarios se realizará sobre la API REST con la que cuenta el proyecto del *servicio de autentificación* [2]. Por otro lado, para el desarrollo del sistema de administración de identidades basado en *blockchain*, en primera instancia, se va a utilizar *Ethereum* como tecnología de cadena de bloques. Se ha decido elegir *Ethereum* por las posibilidades que ofrece para el desarrollo de aplicaciones descentralizadas.

Ethereum es una plataforma *OpenSource*, pero realmente su potencial radica en el hecho de que es programable [3] [4], es decir, permite de una manera sencilla y cómoda desarrollar aplicaciones descentralizadas, las cuales se benefician de la tecnología de las criptomonedas, así como de la tecnología de la cadena de bloques.

La manera en la que podemos programar sobre *Ethereum* es mediante el desarrollo de *Smart Contracts* o *Contratos Inteligentes*. Gracias a estos, se pueden crear aplicaciones descentralizadas, por ejemplo, una aplicación móvil, la cual al interactuar con un *Smart Contract* puede llevar acabo algunas de sus funciones, un buen ejemplo de este tipo de

aplicaciones, podría ser una aplicación de Android la cual te permita realizar encuestas, de manera segura y privada, haciendo uso de esta tecnología.

Para el desarrollo del administrador de identidades, haremos uso de los *Smart Contracts*, los cuales se van a escribir utilizando el lenguaje de programación de alto nivel *Solidity*, el cual sigue la metodología de *diseño por contrato* para crear los contratos inteligentes [3] [4].

Una vez desarrollado el contrato inteligente, se desarrollará la administración de este mediante una API REST, partiendo de la que se desarrolló para el proyecto *Desarrollo de un servicio de autentificación múltiple* [2].

Como ya se ha comentado, este proyecto parte de un proyecto anterior y construye sobre el mismo nuevos desarrollos con lo que alcanzar nuevos objetivos entorno a la identidad digital y la tecnología *blockchain*, proponiendo un modelo de negocio y desarrollando un nuevo sistema que permita conseguir un buen rendimiento y garantizar la seguridad del servicio ofrecido.

3. Estado del arte

Con la solución planteada, se pretende conseguir que nuestro sistema de autentificación contenga un sistema de administración de identidad distribuido, de manera que podamos lograr la delegación de este proceso de otorgación de permisos y roles, en los usuarios de la red de la cadena de bloques en los cuales confiamos.

De esta manera conseguiremos un sistema *Single Sign-On* robusto y confiable, gracias a la tecnología de la cadena de bloques, en la cual se almacenarán de manera ordenada e inmutable registros de autorización, así como roles, permisos y características, que pueden ser comunes a servicios. Consiguiendo de esta manera más privacidad y soberanía sobre los datos del usuario.

3.1 Blockchain

Blockchain o *cadena de bloques* es una estructura de datos en la que la información está contenida y agrupada en un conjunto, llamado bloque, a los que se le añade metainformación relativa a un bloque anterior de la cadena en la línea temporal. De manera, que haciendo uso de técnicas criptográficas la información contenida en cada bloque solo puede ser repudiada o editada modificando todos los bloques que conforman la cadena [5] [6].

Esta propiedad permite que se pueda utilizar como una estructura de datos, la cual puede llegar a ejercer de una base de datos, distribuida, pública, no relaciona y que además contenga un histórico irrefutable de información.

Todo esto ha permitido que mediante criptografía asimétrica y funciones *hash*, la implementación de un registro contable distribuido, al cual se le refiere con el termino *lefger* o *libro de cuentas*, el cual permite soportar y garantizar la seguridad de los datos. Con los protocolos apropiados se puede llegar a un consenso para garantizar la integridad de todas las operaciones realizadas, mediante la participación de todos los integrantes de la red sin tener que recurrir a entidad de confianza que centralice la información.

Por lo que una vez que se ha aprobado bajo consenso el nuevo bloque de la cadena, este se añade a cadena de bloques y esta se sincroniza entre todos los usuarios de la red. De

esta manera toda la información almacenada en el bloque no se podrá borrar ya que son registros persistentes.

Para poder realizar un ataque a este tipo de red, un atacante requeriría de una mayor potencia de cómputo y presencia en la red que el resultante de la suma de todos los restantes nodos combinados.

Por todas las razones anteriores, la tecnología *blockchain* es especialmente adecuada para escenarios en los que se requiera almacenar de forma creciente datos ordenados en el tiempo, sin posibilidad de modificación ni revisión y cuya confianza pretenda ser distribuida en lugar de residir en una sola entidad certificadora.

Cada *red blockchain* tiene sus características, pero todas tienen características comunes. En todas, los bloques que componen la cadena son de un tamaño máximo y un número que indicará el desafío que se debe resolver cada vez, para poder unir el bloque a la cadena. El bloque con un tamaño máximo se utilizará para almacenar transacciones hasta que se alcance dicho valor. Además, cada bloque tendrá una cabecera en la cual se almacenará el *hash* del bloque anterior, de modo que dicho *hash* actúe como puntero o enlace al bloque anterior, formando así la cadena, en la cabecera también se indica la fecha de creación de dicho bloque.

El desafío que se ha de resolver para poder unir el bloque a la cadena es definido como la búsqueda un *hash* que cumpla con las condiciones impuesta por la red de la cadena de bloques.

Es en este punto en el flujo de la cadena de bloques, donde normalmente entran en juego las criptomonedas. Estas criptomonedas son una recompensa para los usuarios que han realizado los duros trabajos y cálculos necesarios para resolver el desafío impuesto, ya que el que el usuario que consiga resolver este desafío será el encargado de unir el bloque a la cadena. La resolución del desafío es aleatoria, ya que para el cálculo de *hashes* cualquiera puede encontrar la solución al desafío, pero cuanta más capacidad de cómputo poseas en la red, más rápido serás a la hora de realizar los cálculos, y por lo tanto tendrás más probabilidades de encontrar la solución, ya que se probarán más combinaciones en un menor tiempo.

Un problema técnico que se plantea en este caso es el famoso problema del *ataque del 51%*. Este ataque consiste, en que un usuario o grupo de usuarios, tengan un poder

computacional del 51% sobre todos los usuarios de la red. De esta forma, se lograría falsear transacciones y bloques, ya que, a la hora de escribir un nuevo bloque en la cadena, se podría indicar lo que ese usuario o grupo de usuarios quisiera en el bloque. De esta forma, como poseen el 51% del cómputo, esa verdad manda sobre lo que se está uniendo a la cadena, frente al 49% de los usuarios que intentan decir que no es cierto. Debido a la complejidad del ataque, es un ataque teórico y no se puede llevar a la práctica en el momento en el que se está escribiendo este documento.

Blockchain ha sido una tecnología que ha explotado, y a la cual se han sumado miles de ideas que intentan explotar las bondades de esta tecnología la cual parece que tiene un alcance ilimitado. En el caso de este proyecto, ya se ha comentado que el interés o la idea que más nos interesa para este proyecto, es la identidad digital.

En el ámbito de la identidad digital, la cadena de bloques podría proporcionar un sistema único, con el que validar identidades de forma irrefutable, segura e inmutable. En este punto podemos encontrar fundaciones como *Sovrin* [7] en la cual se persigue el desarrollo de un mecanismo de identidad digital autogestionada a través de la tecnología de *blockchain* como un estándar, en este caso la fundación utiliza como base la plataforma de código abierto *Hyperledger* [8].

3.2 Ethereum

Ethereum es una plataforma *OpenSource*, descentralizada y programable, lo que quiere decir que, en ella, se pueden desarrollar aplicaciones descentralizadas, también conocidas por el termino *dapps*. Las aplicaciones desarrolladas en *Ethereum* suelen beneficiarse de las criptomonedas, además una vez se han cargado en la cadena de bloques, su ejecución será siempre la misma, ya que al ejecutarse en la cadena de bloques se beneficia del hecho de que es inmutable.

El proyecto de *Ethereum* busca descentralizar la web, en busca de la web 3.0, mediante la introducción de cuatro componentes: publicación de contenido estático, mensajes dinámicos, transacciones confiables y una interfaz de usuario integrada y funcional. Estos componentes están pensados para conseguir una red descentralizada y anónima [3] [4].

Ethereum funciona gracias a la utilización de una máquina virtual, la cual es llamada por sus siglas *EVM* (Ethereum Virtual Machine). En la máquina se ejecutará código *bytecode*, el cual es un archivo binario, el cual contiene un programa ejecutable muy parecido a un módulo, el cual es utilizado por el compilador [9].

Por otro lado, los programas que son realizados mediante contratos inteligentes son escritos mediante lenguajes de alto nivel y haciendo uso de la metodología de diseño por contrato. De esta manera, se consigue crear contratos inteligentes de manera muy similar a como ocurren en el lenguaje de programación *Eiffel* [3] [4].

Los beneficios que presenta la plataforma de Ethereum son los mismos que podríamos encontrar en otras plataformas de *blockchain*, por ejemplo, es inmune a la intervención de terceros en las aplicaciones descentralizadas que han sido desplegadas. Sigue siendo una red de consenso, lo que quiere decir que un cambio involucra todos los nodos de la red. Y, por último, es descentralizada, por lo que hay múltiples puntos de fallo, lo que lo hace más difícil posibles ataques a la red.

Como mayor desventaja presentada por Ethereum, es la vez una de sus mayores ventajas frente a otras plataformas de blockchain, los *smart contracts*. Los contratos inteligentes han de ser escritos por programadores, ya que en el fondo es un programa. El hecho de que una persona tenga que escribir el código implica que puede haber errores humanos, lo que puede probar que estos fallos puedan ser explotados con un mal fin. Si se introducen errores de programación en los contratos, en primera instancia no se podrían detener los ataques, aunque existen formas de poder invalidar un *smart contract* que ha sido desplegado.

3.3 Smart Contracts

En este apartado se va a analizar con mayor profundidad el concepto de *smart contract* y cómo funciona.

Un *smart contract* o *contrato inteligente*, es en resumen un programa informático, el cual hace cumplir y ejecuta acuerdos registrados en entre dos o más partes, especificando lo que se puede hacer, como se ha de hacer o que pasa si no se hace, además, ciertas acciones sucederán como resultado de que se cumplan una serie de condiciones

específicas [10] [11], al igual que ocurriría con un contrato escrito y firmado por dos partes ante notario.

Tienen como objetivo brindar una seguridad superior a la ley de contrato tradicional y reducir los costes de transacción asociados a la contratación. Esto provoca que se traslade el valor a un sistema en el cual no se requiere de la confianza de un tercero.

Al final el contrato vive en un sistema que no es contralado por ninguna de las partes. Los contratos inteligentes no necesitan la intervención o la interpretación humana para llevarse a cabo. Un contrato se ejecutará de manera automática de manera que para una respuesta provoca una acción. Cada nodo en la red actúa como registro de propiedad y de garantía, de modo que se ejecutan los cambios en el contrato y se comprueban automáticamente las reglas que impone la transacción, al mismo modo que comprueba que el resto de los nodos realizan el mismo trabajo.

3.4 Estándares utilizados.

En este apartado se describirán los estándares que vamos a utilizar a lo largo del desarrollo de este proyecto. Junto a estos estándares, se definirán que tipo de medidas de seguridad son necesarias para estos estándares se puedan llevar a cabo.

3.4.1 Cifrado Web

Dentro del cifrado web, nos encontraremos con el protocolo TLS (Transport Layer Security). Este protocolo es utilizado para proporcionar seguridad en la capa de transporte, es decir que se utilizan para las comunicaciones seguras en una red, normalmente Internet.

Cuando decimos comunicación segura, nos referimos a que es una comunicación cifrada, para poder realizar este cifrado se hace uso de certificados *X.509,* y por lo tanto de criptografía asimétrica de modo que el cliente y el servidor negocian el algoritmo de cifrado y las claves que posteriormente se usarán para cifrar el flujo de datos de ambas partes.

Una de las mejores propiedades de este protocolo, es la llamada *forward secrecy*. Esta propiedad, dicta que, conociendo una clave antigua o actual, no puedas saber cuáles

van a ser las que se generen el futuro. De esta manera se dice que un sistema con esta propiedad es un sistema *forward-secure* (sistema seguro-adelante).

Junto a este protocolo de cifrado web, encontramos otro protocolo que hace uso de TLS para mantener nuestras comunicaciones seguras, el protocolo HTTPS. HTTPS (Hypertext Transfer Protocol Secure) es un protocolo de aplicación basado en HTTP, cuyo fin es el mismo, pero haciendo una transferencia segura de hipertexto, es la versión con seguridad de HTTP al usar TLS para crear un canal cifrado.

Usando este protocolo, logramos que, si estamos transfiriendo información sensible a través de la red y alguien logra interceptar nuestra transferencia, no obtendrá la información en texto plano, si no, que lo que obtendrá será unos datos cifrados. Y este es el punto que verdaderamente nos importa para el desarrollo de nuestro proyecto, ya que tola la comunicación que realicemos con el sistema será utilizando este protocolo, para garantizar que la comunicación es realmente segura.

3.4.2 Single Sign On

El *single sign on* o *inicio de sesión único*, es un procedimiento de autenticación que habilita a un usuario determinado para acceder a varios sistemas con una solo instancia de identificación [12], es por ejemplo el principio que podemos encontrar en los *active directory* (directorios activos) de Microsoft, para una red de ordenadores.

En el caso de contar con un sistema de autentificación *single sign on*, hace que el usuario que va a utilizar el sistema tenga que recordad múltiples contraseñas, haciendo el trabajo del usuario más sencillo. Dentro de la tecnología *Single sign on,* existen varios tipos de aproximaciones, en nuestro caso vamos a desarrollar una aproximación entre el tipo *Web* y el tipo *OpenID*.

El tipo *Web* trabaja solo con aplicaciones web, y el principio es el mismo, autenticar a los usuarios en diversas aplicaciones, sin necesidad de volver a autenticar. Los accesos son interceptados con la ayuda de un servidor proxy o de un componente instalado en el servidor web o en la aplicación web destino.

En cambio, el tipo *OpenId*, es un *single sign on* distribuido y descentralizado, donde la identidad de compila en lo que se conoce como una URL, que cualquier aplicación o servidor puede comprobar [12].

3.4.2 JSON Web Token

El estándar abierto JSON Web Token, también conocido por JWT, es un estándar basado en JSON para la creación de *Access Token* o *tokens de acceso*, con los que se permite propagar a otros servicios o sistemas información, datos como la identidad y los privilegios de un usuario, como si fuera un objecto JSON [13].

El JWT se compone de tres partes, la primera *header* que es donde se alberga el algoritmo que ha usado para generar la firma del token. La segunda parte se denomina *payload*, y en ella es donde suele ir toda la información que contiene el token, normalmente utilizando sus propiedades o *claims*.

Y, por último, encontramos la que más nos atañe a nosotros, es la parte donde podemos encontrar la firma. Esta firma está compuesta por las demás partes, es decir el *header* y el *payload*, ambas partes codificadas en *base64*, todo ello cifrado con una clave secreta que estará almacenada en el servidor. El JWT puede ser firmado usando un secreto con un algoritmo HMAC o con una pareja de clave pública y privada usando RSA y ECDSA.

La finalidad que firmar un JWT con una pareja de clave pública y privada, es el poder demostrar que solo la compañía que posee la clave privada ha sido capaz de firmar el token.

Una de las ocasiones en las que deberíamos de usar JWT, es cuando tenemos un cliente que interactúa con nuestro servicio a través de una API REST. En este tipo de APIs el servidor no debe de almacenar ninguna clase de sesión.

El uso de JWT está ligado a las APIs, debido a que las APIs hoy en día son usadas también por clientes HTTP muchos más simples, que no soportan cookies de forma nativa. En muchas ocasiones de se recurre a los tokens para el desarrollo de aplicaciones nativas en el sistema *Android* e *iOS*, ya que en ocasiones en estos sistemas puede haber problemas con el uso de cookies, aunque con cada actualización de ambos sistemas, estos problemas se han resuelto.

Al igual que ocurre con JWT, si solo se va a hacer uso del navegador web como cliente de un servicio, entonces, tiene más sentido utilizar cookies, ya que estos tienen soporte completo para estas.

Probablemente la mayor ventaja de usar tokens y no cookies es el hecho de que ofrecen una autenticación sin estado. Desde *backend* no se necesita tener un registro de los tokens. Cada token es autónomo: contienen en sí mismos toda la data necesaria para confirmar su validez. De esta forma, el único trabajo del servidor es: firmar tokens ante un inicio de sesión exitoso, y verificar que los tokens entrantes sean válidos.

Otra de las situaciones en las que usaría JWT frente a otro tipo de tecnologías, es cuando por ejemplo necesitamos que el tamaño de las peticiones sea lo más pequeña posible, ya que al estar basado en JSON y este ser menos tipado o carece de verbos, el tamaño de este se reduce si lo comparamos por ejemplo con SAML. Es una buena elección si plantemos enviar información a través de HTTP.

Es por todas estas razones, por lo que, en el desarrollo de este proyecto, se van a utilizar los JWT, ya que en este proyecto estamos generando una API REST, y encaja en el modelo descrito anteriormente. Se utilizará para poder mantener la sesión de una manera fluida con el cliente una vez el usuario haya sido autenticado, ya que si tuviéramos que delegar este proceso siempre a la *blockchain* sería un sistema muy lento.

3.4.3 Argon2

Argon2, es una función de derivación de clave ganadora de la competición *Password Hashing Competition* en julio del 2015 [14].

La competición *Password Hashing Competition*, es una competición abierta anunciada en 2013 para poder encontrar y seleccionar funciones de derivación de claves que pudieran ser reconocidas como estándares. Fue organizada por criptógrafos y facultativos de la seguridad informática [15] [16].

En concreto podemos decir que las funciones de derivación de claves, conocidas en inglés por sus siglas KDF, son funciones las cuales a partir de un secreto derivan una o más claves utilizando una función pseudoaleatoria.

Para este proyecto se utilizarán para lo que son utilizadas normalmente, es decir extender contraseñas, así como para darles formato. Con ello, se utilizarán los resultados de estas funciones para comparar los hashes de las contraseñas dentro del flujo de autenticación de usuarios.

En la derivación de clave, hay que tener en cuenta más factores, en el caso de Argon2, deberemos tener en cuenta lo siguiente:

- *Memoria*: la cantidad de memoria que va a usar el algoritmo, en kibibytes.
- *Iteraciones*: el número de iteraciones sobre la memoria.
- *Threads* o *hilos*: número de hilos en la paralelización del algoritmo.
- Tamaño de la *sal criptográfica*: tamaño para la *sal* (sal criptográfica).
- *Tamaño de clave*: tamaño para la clave generada, es decir tamaño de salida del hash.

Todos estos parámetros, son definidos con la intención de prevenir y dificultar un ataque de fuerza bruta. La dificultad de un ataque por fuerza bruta aumenta con el número de iteraciones. En resumen, Argon2 trata de crear un llenado de memoria usando varias unidades de cómputo, explotando así mejor la memoria cache.

Existen tres funciones con tres versiones del algoritmo de *Argon2*:

Argon2d: donde se maximiza la resistencia contra ataques por GPUs.

Argon2i: maximiza la resistencia contra ataques por canal lateral.

Argon2id: es una función hibrida entra las dos versiones de arriba. La primera fase del algoritmo se hace con la aproximación de Argon2i y el resto de algoritmo con la aproximación de la versión Argon2d.

En la documentación del algoritmo de Argon2, se recomienda utilizar siempre la versión Argon2id, y el resto de las versiones de algoritmos solo cuando haya razones de peso para ser utilizadas, ya que pueden presentar inconvenientes frente algunos tipos de ataques [16].

El uso de Argon2 en el desarrollo de este proyecto está justificado por ser uno de los algoritmos de claves más recientes, pero sin duda, las funciones de derivación de clave más conocidas y de las más usadas son *Bcrypt* y *PBKDF2*.

Bcrypt está basada en la función de cifrado *Blowfish*, la cual fue lanzada en 1999, esta función incorpora el uso de una *sal criptográfica* para prevenir ataques mediante *rainbow tables* y hasta la fecha no tiene vulnerabilidades conocidas [17].

Pbkdf2 fue lanzada en el 2000 y cuenta con un coste computacional variable basado en el número de rondas que hace el algoritmo, para poder defenderse de ataques por fuerza bruta, además, también incorpora sal para evitar ataques basado en *rainbow tables*. Hasta la fecha tampoco cuenta con vulnerabilidades conocidas [18].

Argon2 es del 2015, es decir es 15 años más joven que las otras funciones conocidas de derivación de clave. Este hecho, puede ser bueno y malo, bueno porque ha podido formase contra los ataques más utilizados y conocidos en esos 15 años de diferencia, y malo, porque puede que tenga debilidades en el algoritmo que aún no han sido descubiertas y en esos 15 años el resto de las funciones han tenido tiempo de ser más estudiadas.

4. Planificación del proyecto

A continuación, vamos a explicar que metodología se va a utilizar para el desarrollo de este proyecto, así como la planificación del proyecto. Una vez se haya definido la metodología a emplear, también se definirá el entorno ajo el que se ha realizado el desarrollo y las pruebas.

En este apartado se analizarán además los posibles casos de negocio de los que dispone este proyecto, pese a que se utilizará de base un proyecto de código abierto y por lo tanto este proyecto también lo será.

4.1 Metodología de desarrollo

La metodología de desarrollo software es un marco de trabajo usado para estructurar, planificar y controlar el proceso de un desarrollo de software [19].

Podemos encontrar múltiples metodologías de desarrollo, cada una con sus fortalezas y sus debilidades.

Durante el desarrollo de este proyecto se utilizará como metodología de desarrollo el modelo iterativo e incremental. El enfoque de este modelo nos permite dividir el desarrollo en iteraciones de tiempo, de manera que podemos conseguir versiones estables del producto final, con cada iteración. En el modelo iterativo e incremental, se busca evolucionar el producto de manera que cada versión se apoya en las anteriores para mejorar hasta la próxima iteración, es una manera de retroalimentación para ampliar las funcionalidades del producto.

Las claves para un correcto funcionamiento de este modelo son que hay que saber priorizar los objetivos en función del valor que aportan estos al producto final y entre los que se consideran funcionalidades esenciales para este.

Debido a que el proyecto sobre el que se desarrollará este producto ya fue diseñado con este tipo de metodología, hace que sea más sencillo el desarrollo de este, ya que podemos considerar el desarrollo de este producto, como una iteración más sobre el proyecto base, de manera que varios o muchos de los aspectos ya desarrollados y probados pueden utilizarse para desarrollar este producto.

Esto no solo hace el desarrollo sea más ágil, sino que también hace que el producto a desarrollar sea más sólido y con menos tasa de fallos o propenso a fallos, ya que los aspectos que se reutilicen a habrán sido probados y contendrán métodos seguros para el tratamiento de la seguridad y privacidad de los datos del usuario.

En un principio se había pactado con el profesor reuniones presenciales para poder definir bien los requerimientos, resolver dudas y determinar el alcancé del proyecto. Dada la situación actual, con la pandemia y la cuarentena debida al COVID-19, se han cambiado las reuniones presenciales, por reuniones telemáticas, es decir mediante video conferencia, estas reuniones tienen la misma finalidad que las reuniones presenciales, además se cuenta con el correo electrónico como otro canal para poder realizar dicha comunicación.

Con el procedimiento establecido, se pretende llegar a una versión sencilla, pero funcional del producto, ya que este desarrollo puede ampliarse mucho hasta alcanzar lo que se consideraría un producto final.

4.2 Entorno y tecnologías de desarrollo

Para el desarrollo de este proyecto nos tendremos que apoyar en diferentes tecnologías de desarrollo para crear *smart contracts*, así como para interactuar con la *red blockchain*, que en este caso recordemos que es Ethereum. Es por ello, por lo que el smart contract será desarrollado con el lenguaje de Programación Solidity.

Para este proyecto, además, se va a desarrollar una API REST escrita mediante el lenguaje de programación *Go*, la cual se utilizará para interactuará con el servicio,

Durante el desarrollo se utilizará un ordenador, el cual contará con el sistema operativo *Windows 10*, además, también se contará con una máquina virtual con el sistema operativo *Ubuntu 19.04*, la cual será utilizada para instalar el compilador de *Solidity* y usarlo en el desarrollo del *smart contract*.

4.2.1 API (Application Programming Interface)

Una API es un conjunto de subrutinas, funciones y procedimientos que ofrecen una librería de software para que estas puedan ser utilizadas por otro software.

En este proyecto se desarrollará una API, de tipo API REST. El estilo definido por una API REST, hace referencia a un conjunto de principios de arquitectura en la creación de aplicaciones.

Los principios que definen este estilo son los siguientes [20]:

- Un protocolo cliente / servidor sin estado. Todos los mensajes HTTP enviados deben tener toda la información necesaria para realizar la petición de manera correcta según su ámbito. De esta manera ni el cliente, ni el servidor deben almacenarán el estado o sesión del usuario.
- Operaciones bien definidas en lo que ha operaciones HTTP se refiere. Esto quiere decir que utilizan los verbos HTTP, a la vez que se hace un buen uso de las operaciones de HTTP, como pueden ser POST, GET, PUT o DELET.
- Sintaxis universal. Cada recurso es accesible desde una URI única.
- Formato XML para cada mensaje y para cada uno de los recursos.

Aunque esos son los principios que definen el estilo REST, en la actualidad, el termino se utiliza para definir cualquier interfaz ente sistemas que utiliza HTTP para obtener datos o para ejecutar opciones sobre dichos datos. Normalmente utilizado también para referiste a una interfaz, la cual utiliza el formato JSON para el intercambio de mensajes.

4.2.2 Go

Go es un lenguaje de programación desarrollado por Google, que salió al mercado en 2009. Es un lenguaje de programación concurrente, compilado y orientado a objectos, con una sintaxis muy similar al lenguaje de programación C, este hecho es debido a que en el desarrollo de *Go*, está presente Ken Thompson, el cual es el creador de UNIX y del lenguaje de programación B, el cual fue el lenguaje precursor del lenguaje *C*, pero con cada versión de *Go*, este se separa más y más del lenguaje de programación *C* [21] [22].

Go está disponible para todos los sistemas operativos del mundo, incluido Windows, que es sobre el que se realizará el desarrollo de este proyecto. Esto es debido a que contamos con el lenguaje en formato binario, por lo que nos permite instalarlo en cualquier sistema.

Go recibe su inspiración de muchos de los lenguajes de programación que había en el mercado, ya que cuenta un gran rendimiento en red y multiprocesos, así como un lenguaje tipado estático, lo que le caracteriza con una gran eficiencia en tiempo de ejecución, muy parecido a lo que ocurre en lenguajes como JAVA o C++. Además, cuenta con varias características de legibilidad propias de los lenguajes dinámicos, parecido a lo que ocurre en leguajes como Python o JavaScript.

La mejor característica que podemos encontrar en este lenguaje para el desarrollo de este proyecto es su gran comunidad. Esta comunidad que encontramos detrás del lenguaje hace que cuente con una gran librería estándar con números paquetes para abarcar la mayoría de los algoritmos y funciones necesarias para el desarrollo software actual.

En concreto, durante el desarrollo de este proyecto, se utilizará el paquete *cryto*, que nos permitirá usar los algoritmos y estándares criptográficos usados en este proyecto. Este paquete es uno de los más completos en cuanto a criptográfica se refiere, que podemos encontrar en estos momentos en los lenguajes de programación actuales

Como hemos mencionado antes, *Go* está disponible para Windows. Durante el proceso de instalación, deberemos de definir las variables de entorno que utiliza el lenguaje para navegar por el sistema, dichas variables serán *GOROOT* y *GOPATH*.

La variable de entorno *GOROOT* se utiliza en el proceso de instalación de G. Durante el proceso de instalación *Go* asume que la ruta por defecto será *usr/local/go*, en Linux, o en *C:\go*, en el caso de Windows. Gracias a *GOROOT*, podremos instalar *Go* y todas sus herramientas en una localización diferente.

Con la variable de entorno de *GOPATH*, indicaremos la localización donde *Go* buscará nuestro código escrito en *Go*. En esta localización es donde se añadirán y buscarán los paquetes que no pertenezcan a la librería estándar de *Go*, así como el código que nosotros desarrollemos.

Una vez tengamos *Go* instalado en nuestro ordenador tendremos el entorno listo para poder comenzar a desarrollar la API REST en *Go*. Para ello que utilizaremos *Visual Studio Code* como editor de código fuente, junto al plugin del lenguaje *Go*, oficial de Microsoft para este editor de código.

Visual Studio Code es un editor de código creado por Microsoft, el cual es de código abierto [23]. Visual Studio Code, nos permitirá gracias al plugin, instalar paquetes para *Go*, tanto oficiales como de terceros, autocompletado y coloreado de sintaxis, así como desarrollar, ejecutar y depurar aplicaciones escritas en *Go*, además de ofrecernos un control de versiones mediante Git.

4.2.3 Solidity

Solidity es un lenguaje de programación utilizado para escribir *smart contracts* (*contratos inteligentes*), este lenguaje es utilizado por varias plataformas de *blockchain* para el desarrollo de s*mart contract*, pero sin duda la más popular es *Ethereum*.

Este lenguaje de programación fue propuesto por Gavin Wood en 2014 y creado por Christian Reitwiessner, Alex Beregszaszi, Yoichi Hirai y contribuidores del *core* de *Ethereum*. Además, fue diseñado basándose en la sintaxis del lenguaje *ECMAScript*, para poder definir una sintaxis familiar para los desarrolladores web.

Es un lenguaje tipado y diseñado para el desarrollo de contratos inteligentes, los cuales corren en una *EVM* (Ethereum Virtual Machine). *Solidity* se compila en *bytecode*, de manera que es fácilmente transportable a cualquier entorno en el cual se pueda ejecutar, en este caso una EVM. Mediante este método, los desarrolladores, pueden crear aplicaciones que implementan la lógica de negocio de manera autoejecutable e incorporándola en contratos inteligentes.

Para desarrollar los s*mart contracts*, también utilizaremos el editor de código Visual Studio Code, y al igual que ocurría con *Go*, para poder programar en este lenguaje se hará uso de un plugin. Además, para poder compilar el código, como ya se ha mencionado en el apartado anterior, utilizaremos una máquina virtual con Ubuntu para poder instalar el compilador de *Solidity*.

Por último, se utilizará la web *Remix Ethereum IDE* [24], la cual nos permitirá realizar pruebas de ejecución e interactuación con el contrato, para poder entender el lenguaje, así como permitirnos la depuración en tiempo real de los *smart contracts*, gracias al uso de las máquinas virtuales de JavaScript.

En la web podemos encontrar también documentación y consejos para escribir mejores contratos inteligentes, así como proporcionar herramientas para definir y entender cómo utilizar el *gas* en los *smart contracts*.

4.2.4 Ganache

Con el *framework* Ganache puedes tener tu propia *red blockchain* en tu ordenador de una manera sencilla para poder realizar pruebas y desarrollar para *Ethereum* [25].

Ganache simula una *red blockchain* de *Ethereum* con todas sus características y forma parte de la suite de herramientas *Truffle Suite*. Esta *suite* nos ofrece herramientas, mediante las cuales podemos recrear entornos de *blockchain* para crear y probar *smart contracts*, ya que este tipo de pruebas y ejecuciones difiere mucho de otros desarrollos software.

Este tipo de herramientas han sido diseñadas porque el desarrollo de los *smart contracts* es complejo si lo intentamos hacer directamente sobre la red de *blockchain* pública. Durante el desarrollo de s*mart contracts*, hay que tener en cuanta muchas cosas, primero que las transacciones solo se pueden comunicar con la *red blockchain*, segundo, que las interacciones con el *smart contract* son asíncronas, es decir que los efectos de la transacción no pueden ser vistos hasta que no se hayan escrito en el bloque correspondiente de la cadena. Y, por último, se debe tener en cuenta que la cadena de bloques impone restricciones sobre el código que tiene que ejecutar, como, por ejemplo, el límite de gas que se permite gastar por operación o ejecución de funciones.

Cada despliegue de un *smart contract* en la red tiene un coste asociado, por lo que no se puede permitir tener que desplegar muchas veces el mismo contrato para poder realizar pruebas. Es por estas razones por las que aparecen herramientas como *Ganache*.

Ganache está disponible tanto en un entorno gráfico, como por línea de comandos, además de que puede ser instalado como un paquete *NPM*. Una vez descargado e instalado, con tan solo un clic o un comando, tendremos nuestra propia *red blockchain* en local.

Una vez arrancada nuestra red de cadena de bloques, Ganache nos generará 10 cuentas de usuarios para poder interactuar con los *smart contract*, así como proporcionarnos un explorador de bloques y transacciones transmitidos a la red y un *log* para la comprobación de nuestro sistema.

En el explorador podremos ver los bloques, de los cuales se pueden consultar la fecha en la que fue minado, el número de transacciones que recoge, el gas utilizado, el hash del bloque y que dirección que realizó la transacción. Dentro del información detallada de las transacciones se pueden consultar el *hash* de la transacción, la dirección de *Ethereum* que ejecuto, el tipo de transacción, el *gas* utilizado y si además la transacción es un despliegue de un *smart contract*, podremos ver la dirección de dicho contrato.

4.2.5 SQL Server

SQL Sever, es el gestor de base de datos relacionales diseñado desarrollado por Microsoft. Siempre había estado disponible en exclusiva para sistemas operativos Windows, pero desde 2016 está disponible para sistemas GNU/Linux y desde 2017 para Docker [26].

En este gestor de bases de datos, se utiliza un lenguaje propio derivado de SQL, llamado T-SQL. Esto cambio en el lenguaje, hace difícil el cambio si en algún momento queremos comenzar a utilizar otro gestor de bases de datos, ya que si utilizamos características del lenguaje T-SQL tras un cambio de gestor de base de datos no podremos hacer uso de las mismas. Es por ello, que para poder dejar el código lo más sencillo y manejable posible ante futuros cambios, por lo que no se van a utilizar ninguna característica de este lenguaje.

En este aspecto supone una mejora frente al proyecto de partida, ya que en este proyecto sí que se utilizaba la característica de los *schemas* propios de este lenguaje y gestor de bases de datos.

Para poder realizar la conexión a las bases de datos de SQL Server con el lenguaje de programación *Go*, utilizaremos el *driver* llamado *go-mssqldb* [27]. El único requisito para poder utilizar este driver con *Go*, es utilizar la versión 1.8 o superior del lenguaje.

4.2.6 Postman

Postman es un software especializado en el concepto de *API Testing*. Este software nos permitirá realizar pruebas sobre la API que desarrollemos. Estas pruebas podrán ser de manera directa como parte de los *tests* de integración, para comprobar que la API cumple

con requisitos en cuanto a funcionalidad, rendimiento, integridad, confidencialidad y disponibilidad. Las APIs, deben de ser probada mediante el intercambio de mensajes.

Postman, actúa como un cliente HTTP, lo cual nos permitirá poder realizar pruebas para APIs. Postman, nos brinda la oportunidad de crear baterías de pruebas automatizadas para poder comprobar en cada versión o cambios realizados, se cumplen todos los requisitos previstos. A esta funcionalidad, le acompaña otra mediante la cual podemos tener todas las llamadas clasificada en colecciones, para tener toda la información de la API organizada para cada una de las rutas de la misma.

Por último, una gran ventaja que posee Postman, son las variables de entorno, las cuales nos dejaran definir variables comunes para una colección de llamadas, por ejemplo podemos tener como variable de entorno, la IP del servidor al que se va a realizar la solicitud, de esta manera conseguimos poder tener dos variables de entorno, una apuntando a nuestra API en desarrollo y otra apuntando a nuestra API en producción, de manera que cambiando las variables activas, podemos realizar la misma prueba contra APIs diferentes.

Como ya he mencionado antes, la utilización de Postman en este proyecto a servirá para poder probar la API de una manera cómoda y automatizada.

4.3 Modelo de negocio

Como veremos en el apartado de implementación, el punto de partida es el proyecto llamada *Desarrollo de un servicio de autentificación d factor múltiple*, el cual es un proyecto que tiene una licencia GNU (General Public License) también conocida por sus siglas en ingles GNU GPL o GLP para abreviar [28].

Esta licencia es ampliamente utilizada en el mundo del software libre, pues garantiza a los usuarios la libertad de poder estudiar, utilizar y modificar el software en la manera en la que se desee. Pero a su vez, esta licencia, impone unas restricciones para proteger al software que contiene esta licencia. La restricción es que no se pueden modificar las libertades nombradas anteriormente, cuando se realicen procesos que impliquen distribución o modificación del software. De esta manera, todo el software derivado de un software con esta licencia automáticamente debe de llevar la misma licencia, para proteger las libertades del software.

Siguiendo con la licencia GNU, todo el código fuente del proyecto lo podemos encontrar almacenado en GitHub [29]. Dentro del repositorio podremos encontrar la API organizada entorno a los recursos, así como la generación de tokens JWT y los diferentes *smart contracts*.

Debido a las características de *blockchain*, no se podrá tener como modelo de negocio un servicio privado para el usuario, como se verá más adelante en el apartado de implementación.

4.3.1 Servicio comercial para empresas

Como ya hemos visto el proyecto será *Open Source,* pero no por ello tendremos menos oportunidades de negocio. En el modelo de servicio comercial para empresas, nos referimos a un modelo de negocio parecido al que tiene Canonical.

Este modelo de negocio consiste en distribuir el software de manera libre y diseñaríamos un modelo de negocio, en el cual ofrecemos un servicio de administración y configuración del mismo servicio, normalmente dirigido a empresas.

En este modelo de negocio, el cliente tiene la posibilidad de elegir si ellos mismos quieren configurar el servicio o si por el contrario contratan al servicio de administración que creó el servicio.

De esta manera el cliente estaría contratando una solución integral, oficial y de confianza, la cual asegurará que se cumplen todos los requisitos de seguridad y que se cubren los principios de integridad, disponibilidad y confidencialidad. Además, se otorgaría un servicio de asistencia y servicio técnico 24/7, generación de informes y mantenimientos.

4.3.2 Donaciones

Este sería el modelo de negocio, con que el que menos llegaríamos a lucrarnos ya que varía mucho en función de si a la gente le ha gustado el proyecto y quiere agradecerlo.

Las donaciones podrían ser a través de transferencias o través de portales, como por ejemplo *BountySource*. El cual, es un portal que conecta proyectos *Open Source* con los usuarios, de manera que estos pueden donar para simplemente apoyar el proyecto, o donar

para cumplir con tareas concretas del propio proyecto, como por ejemplo desarrollar una funcionalidad concreta o corrección de fallos. De manera que el portal hace seguimiento de los *pull request* dentro del repositorio de código del proyecto, y cuando se cumplen dichos objetivos el portal libera el dinero, muy parecido a lo que ocurre con plataformas como *Kickstarter*.

5. Desarrollo del proyecto

Partiendo del proyecto base *Desarrollo de un servicio de autentificación de factor múltiple,* nombrado anteriormente, se realizará modificaciones para transformar la gestión de identidades del servicio en un gestor de identidades basado en *blockchain*.

En este apartado se explicará cómo ha sido utilizado el proyecto base, el cual será el punto de partida, así como el desarrollo de cada una de las partes nuevas, asegurando que todos los procesos que se han seguido son seguros, para así conseguir que el servicio siga siendo un buen sistema de autentificación de usuarios.

A lo largo del desarrollo, podremos observar la importancia que tiene la cadena de bloques sobre el proceso de gestión de identidades, tanto en el ámbito de los clientes que añadan este sistema a su servicio, como en los usuarios que hagan uso del servicio.

Veremos, que muchos de los aspectos que contenía el servicio de autentificación, siguen estando presentes, como por ejemplo los segundos factores de autentificación biométricos. Esto quiere decir, que se reutilizarán elementos del proyecto del cual se parte, ya que no se han eliminado aspectos de seguridad del servicio, sino que se ha amoldado el servicio del proyecto base, al nuevo modelo de negocio y al nuevo gestor de identidades y, en definitiva, se ha integrado la misma seguridad en el nuevo servicio.

5.1 Punto de partida

El proyecto base *Desarrollo seguro de un servicio de autentificación con factor múltiple,* contaba con las siguientes partes:

- API para usuarios.
- API para clientes.
- Una aplicación Android para los usuarios.
- Una aplicación Android para los clientes.
- Página web para los usuarios.

En esta estructura, se planteaban tres modelos de negocio, en los cuales todo el software era *OpenSource*.

El primero de ellos consistía en un servicio privado, en cual los clientes que quisiera usar el sistema de autentificación tenían dos opciones, la primera, era en la que ellos mismo podían descargar de manera libre todo el código alojado en la plataforma *GitHub* y la segunda opción consistía en un servicio privado montado y configurado de manera correcta en un entorno *cloud*.

Por un lado, si el cliente elegía la primera opción y descargaba el software, era responsable de la seguridad del servicio, ya que los propios clientes eran quienes debían de configurar todo el servidor, la API y las bases de datos para la gestión de los usuarios. Si todos estos puntos no eran realizados correctamente, la seguridad del servicio podía llegar a reducirse o incluso llegará a desaparecer.

Por otro lado, si el cliente contrataba el servicio que estaba alojado en *cloud*, estarán contratando el servicio tal y como se diseñó, y con la comodidad de solo tener que darse de alta para poder comenzar a autenticar usuarios en sus sistemas o servicios, muy parecido a lo que ocurre con tecnologías como WordPress. Este modelo, además de proporcionar toda la configuración sin preocupaciones, te ayudaba a autenticar usuarios, sin que debieses tener un servidor propio o un entorno *cloud*.

Además de todo lo nombrado, el sistema privado, contaba con un servicio técnico capaz de resolver cualquier problema que el servicio ofreciese.

En ambos casos, el servicio siempre otorgaba una aplicación para dispositivos inteligentes, ya que era de vital importancia para poder crear un segundo factor de autentificación para el servicio, ya fuera un segundo factor por posesión, como biométrico.

En cuanto a las aplicaciones para dispositivos inteligentes, si no se contrataba el sistema y se optaba por descargarlo y montarlo por su cuenta, el servicio proporcionaba una aplicación genérica, la cual al estar desarrollada con el *framework* de *Xamarin* podía dar soporte a todos los sistemas que el *framework* daba soporte. En cuanto a las posibilidades de selección biométrica, la lista era más limitada, es decir, había menos opciones entre las que elegir para el segundo factor de autenticación, pero dejando opciones igual de válidas para la seguridad.

Si se contrataba el servicio, se proporcionaban aplicaciones personalizadas y todo el gasto de distribución de la aplicación corría de manos del servicio contratado. Además de todo ello, el cliente podía solicitar aplicaciones para sistemas que no estuvieran soportadas

por *Xamarin*, así como hacer peticiones sobre la integración de más sistemas de segundo factor de autentificación.

El software del *servicio de autentificación* era de código abierto con licencia GNU/Linux. Esta licencia nos permite modificar el software siempre y cuando respeten la licencia del software y recordando que todo el software desarrollado a partir de ese proyecto deberá de contener la misma licencia que este.

Para el proyecto del *servicio de autentificación* se eligió este tipo de licencia de código abierto, ya que permitía que un cliente pudiera desarrollar soluciones para un segundo factor de autentificación. En este caso, esa posibilidad sigue estando y se tendrá que fomentar más ya que como veremos más adelante, ahora ya no se puede contemplar un modelo de negocio privado.

Como hemos visto, el software podía seguir siendo mantenido por la comunidad, en el caso de este proyecto, el software base o sobre el que se traza el punto de partida, al derivar el nuevo proyecto de otro con la licencia GNU/Linux, automáticamente, el nuevo proyecto desarrollado pasará a contar con la misma licencia GNU.

Una vez llegados a este punto, ya hemos podido ver cómo es la estructura de la que partimos. Pero el problema que plantea la arquitectura previa es que, si se mantiene el sistema de servicio privado, es imposible implantar el sistema de gestión de identidades basada en *blockchain*.

¿Por qué no se puede mantener el sistema privado junto con el nuevo sistema de administración de identidades basada en *blockchain*? La respuesta a esta pregunta es que, si se realiza mediante un sistema privado, este sistema con *blockchain*, no aporta valor.

Se puede crear un servicio de autentificación privado y utilizar la tecnología de la cadena de bloques para almacenar la identidad del usuario, ya sean roles, permisos o características de los usuarios, pero eso no cambia nada, ya que lo único que cambias es una base de datos tradicional por una base de datos distribuida e inmutable. Pero, además, si se utiliza en un sistema privado, tienes dos opciones, o crear una *blockchain* propia, es decir una *blockchain* privada o utilizar una de las que ya hay en el mercado para llevar a cabo tu cometido, es decir utilizar una *blockchain* pública.

En el caso de utilizar una *blockchain* privada es difícil, no solo por la implementación que ello supone, si no por el mantenimiento y almacenamiento de dicha

blockchain. En el supuesto de que se siguiera adelante con esta implementación, el modelo de negocio tendría que ser diferente, ya que los clientes deberían de asumir el coste y el mantenimiento de la *blockchain*.

Además, estas cadenas de bloques cuentan con un sistema de permisos, es decir es una única entidad la encargada de mantener la cadena, dar permiso de acceso a los usuarios, así como aceptar bloques nuevos de la cadena y proponer las transacciones pertinentes, lo que va un poco en contra con el sistema de confianza de las cadenas de bloques.

Por último, en las *blockchain*s privadas, la base de datos se encuentra almacenada en servidores centrales y no está abiertos al público. Es por todas estas razones por las que la mayoría de los usuarios no consideran a estas redes como *redes blockchain*. Todo esto causa desconfianza y no se puede llegar a saber hasta qué punto la red es transparente y anónima.

Por todas estas razones, no es compatible el uso de una *blockchain* privada con el sistema que de autentificación *single sign-on* que se plantea en este proyecto.

Por otro lado, si utilizamos una *blockchain* de las que podemos encontrar en el mercado, es decir una *red blockchain* pública, no habrá administradores, los usuarios serán realmente anónimos, cualquier persona tendrá acceso, la base de datos será mantenida por todos los usuarios y por ende la base de datos se encuentra almacenada de forma masiva por todos los usuarios, es decir, cada usuario posee una copia exacta de la cadena de bloques.

Todas estas características la hacen idónea para realizar la aproximación que buscamos con nuestra solución, pero si mantenemos el servicio privado con una *red blockchain* pública seguiríamos teniendo la misma problemática que hemos visto con las redes privadas *blockchain*, es decir seguiremos teniendo una única entidad encargada de administrarlo todo.

Es por todo lo que hemos visto, por lo que desaparecen muchos de los elementos que componían el servicio base, para pasar a tener una arquitectura en la que se utiliza una *red blockchain* pública y cuyo fin, es la comunicación entre usuarios en base a la confianza generada por la *red blockchain*.

5.2 Diseño del nuevo servicio

Por todas las razones nombradas en el apartado anterior la arquitectura del servicio base cambiará. La arquitectura previa era la siguiente:

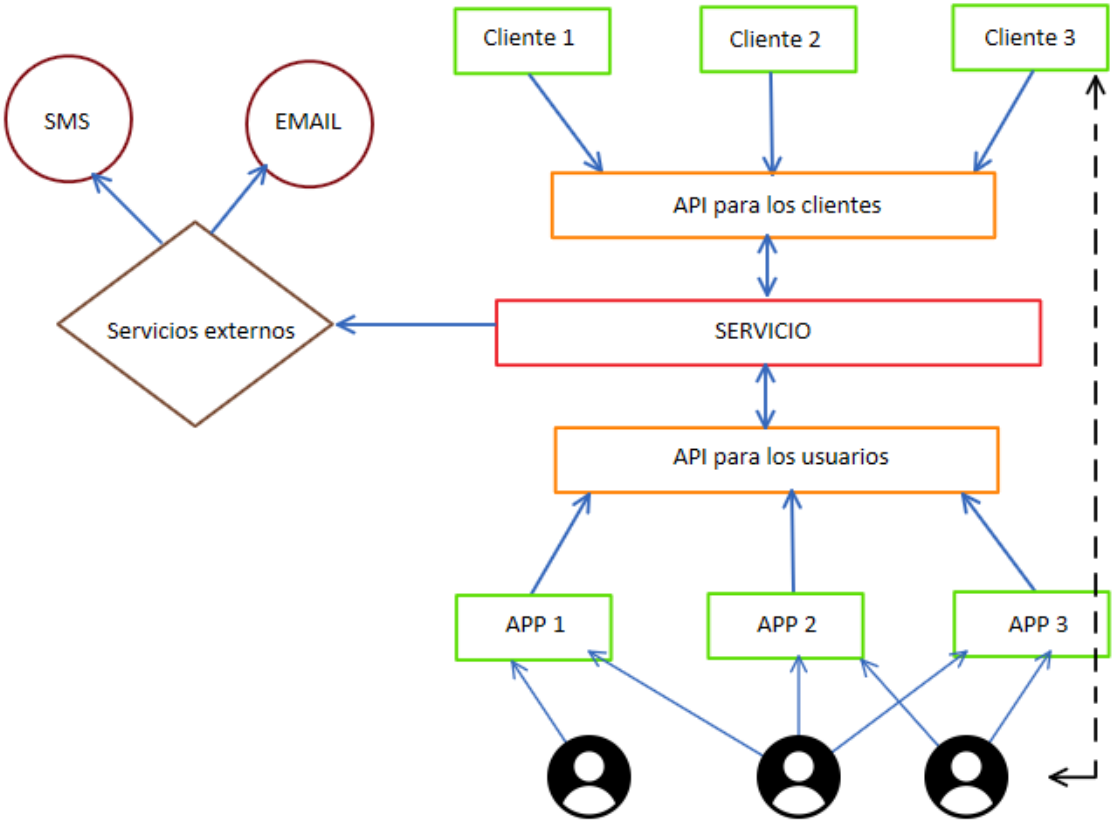

Figura 1 Arquitectura de Desarrollo de un servicio de autentificación de factor múltiple

Y la nueva arquitectura, tendrá el siguiente modelo:

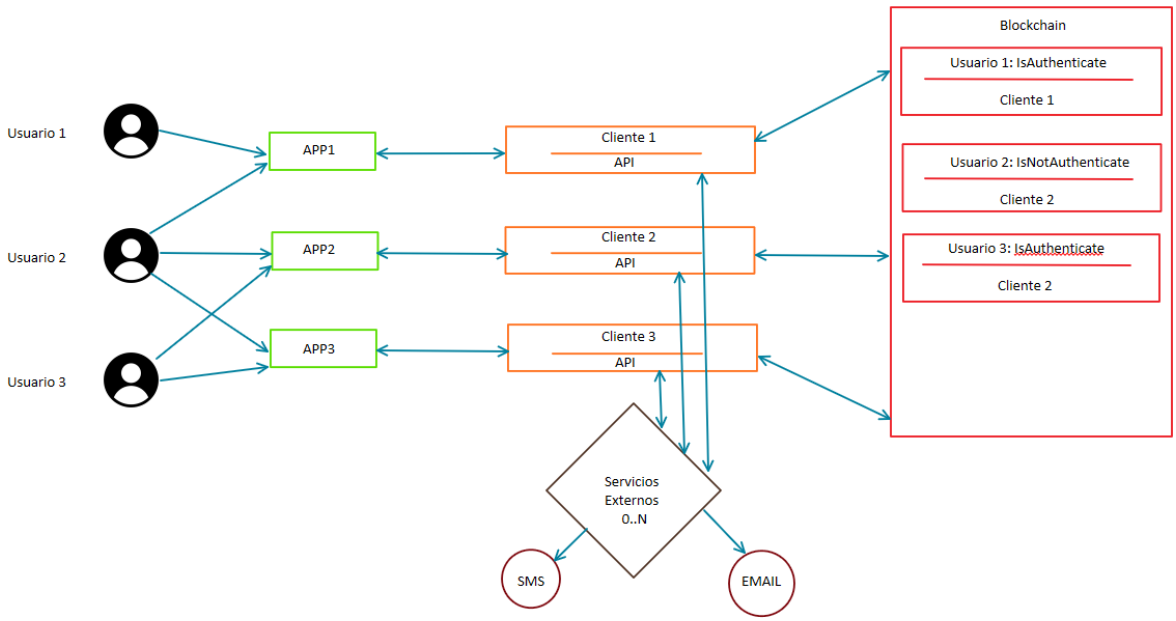

Figura 2 Arquitectura del nuevo sistema

Como se puede apreciar el cambio más apreciable a primera vista, es el hecho de que cada cliente cuenta con una API REST propia, es decir el servicio se ha acercado a los clientes.

Lo primero que desaparece, son las aplicaciones para dispositivo inteligentes. La desaparición de estás es debido a que ya no forman un servicio privado único, si no que cada cliente proporciona un servicio y cuyo sistema de autentificación será el proporcionado por nuestro sistema, junto con un sistema de administración de identidades basadas en *blockchain*. Por lo tanto, será responsabilidad de cada cliente el proporcionar un método con el que interactúa con nuestro sistema de autentificación, mediante llamadas a la API, ya sea una aplicación para dispositivos inteligentes, una aplicación web…

El siguiente elemento en desaparecer es la API REST para los clientes. En la nueva arquitectura, no tiene sentido mantener una API REST cuya finalidad era la de poder realizar gestiones del servicio privado como clientes, gestiones del tipo, darse de alta en el servicio, o añadir y quitar segundos factores de autentificación.

Con la nueva arquitectura el cliente configurará todos los aspectos del sistema de autentificación, para que se ajuste a su servicio, además, como se ha explicado en el párrafo anterior, ya no habrá aplicaciones que consuman dicha API REST.

Por lo tanto, al eliminar la API REST correspondiente con ofrecer un sistema de gestión del propio servicio a sus clientes, solo nos deja la API REST destinada para los usuarios, de la cual hemos visto que todos los clientes tendrán una copia. En esta API REST es donde realmente se lleva a cabo todo el flujo de autentificación de usuarios, así como el manejo de sus identidades.

Otro de los elementos que se mantienen son los servicios externos, pero con unas pequeñas modificaciones. Antes era el servicio de autentificación el encargado de añadir, configurar y subministrar estos servicios externos. Ahora, será la tarea de cada uno de los clientes la de configurar y elegir los servicios externos.

Hasta este punto, ya hemos hablado de todos los elementos que conformaban la arquitectura del servicio sobre el que parte este proyecto, pero en la nueva arquitectura, se ha añadido un nuevo elemento, el cual es el más importante, la *red blockchain*.

La *red blockchain*, será utilizada para gestionar las entidades de los usuarios cuyos clientes estén utilizando nuestro sistema de autentificación. En ella se almacenarán de manera ordenada e inmutable registros de autorización, así como roles, permisos y características, que pueden ser comunes a servicios.

En este caso, la red pública de *blockchain* que vamos a utilizar es, como ya hemos visto con anterioridad, *Ethereum*. Dentro de la *red blockchain*, se utilizarán *smart contracts* para el desempeño de la lógica de negocio, de nuestro sistema de gestión de identidades.

Como se puede apreciar en el diagrama, todos los clientes que utilizan nuestro sistema deberán de utilizar la red de *Ethereum*, ya que el diseño del contrato inteligente será diseñado con el lenguaje de programación *Solidity* exclusivamente para *Ethereum*.

5.3 Implementación del nuevo servicio

En este apartado veremos cómo se han implementado todas y cada una de las partes que conforman el nuevo servicio.

Como hemos visto en el apartado de diseño, un aspecto fundamental del nuevo servicio desarrollado es la integración e interactuación con la *red blockchain Ethereum*. Es por ello por lo que comenzaremos hablando de la implementación de su característica más importante, los *smart contracts*.

5.3.1 Desarrollo del *smart contract*

Para el desarrollo del contrato inteligente, se utilizarán dos herramientas, la primera será *Visual Studio Code* junto con una extensión, la cual nos permite, tanto escribir como interpretar el lenguaje de programación *Solidity*. La segunda herramienta será la web *Remix Ethereum IDE* [24]. Con esta herramienta realizaremos pruebas, para poder definir y entender bien los límites que tiene el contrato inteligente, así como la fantástica funcionalidad de depuración de contratos inteligentes que posee esta web.

A la hora de escribir este documento, se tienen múltiples versiones del *smart contract* tal y como se podrá ver en el repositorio de código. Pero durante todo el desarrollo de la memoria, se hará referencia y se explicará la última versión disponible de este, ya que será la que contará con todas las novedades y todos los arreglos de fallos.

Lo primero que deberemos de saber acerca del desarrollo de *smart contracts* con *Solidity*, es que debemos de especificar, bajo que versión del lenguaje se está definiendo el contrato. Esta especificación es añadida al contrato, parecido a un espacio de nombres, utilizando la directiva *pragma.*

Con cada versión del lenguaje de *Solidity* se cambian muchas cosas, se eliminan otras o se realizan arreglos y mejoras, el mayor cambio y el cual puede que los contratos más antiguos no funcionen con las versiones más nuevas, son los cambios introducidos entre la versión 0.5.0 y la versión 0.6.0 ¿Cuál es el problema con la versión del lenguaje, el compilador y la *blockchain*? El problema es que una vez tengamos un contrato desplegado en la cadena la de bloques, no vamos a poder modificarlo.

Una segunda condición con la versión del lenguaje a utilizar es el símbolo “^” junto con el numero de la versión, de la siguiente manera:

```
pragma solidity ^0.5.2;
```

El símbolo “^” indica que el contrato no compilará con un compilador que utilice una versión más antigua que la versión 0.5.2 y que tampoco compilará con versiones que utilicen la versión 0.6.0 o superior.

La idea que hay detrás de esto es que no habrá cambios que rompan el funcionamiento del contrato hasta la versión 0.6.0, es decir que nuestro contrato compilará de igual forma que lo ha hecho en el desarrollo del mismo. De esta manera la corrección de errores del compilador se mantiene durante toda la vida útil. De esta forma nos aseguramos de que el contrato seguirá funcionando tal y como lo ha hecho mientras lo hemos desarrollado [30] [31].

Usar la versión con la directiva *pragma* no hace que el compilador cambie de versión, ni que este desactive o active funcionalidades para la compilación. Solamente hace que el compilador compruebe la versión antes de comenzar y comparar para ver si posee la misma que le hace falta para compilar, si no coinciden el compilador mostrará un error [30] [31].

Como vemos, el número de versión es muy importante en los s*mart contract* es por ello, por lo que se recomienda siempre leer la lista de cambios de cada versión antes de utilizarla.

En nuestro caso este contrato va a ser desarrollado con la versión 0.6.0 de *Solidity*, la cual no es la última, en el momento en que se está escribiendo este documento, ha salido hace unos días la versión 0.6.8. Todo lo descrito a continuación será sobre la versión 0.6.0, puede cambiar con versiones posteriores o anteriores.

Una vez que hemos definido la versión que vamos a utilizar, comenzamos el desarrollo del contrato con la palabra clave *contract* seguido por el nombre del contrato, esta palabra reservada puede verse como el equivalente a la palabra *class* de otros lenguajes de programación.

Una vez hemos comenzado el desarrollo del contrato se definirán las variables. Pero las variables en *Solidity* funcionan de una manera diferente a los lenguajes de programación convencionales.

Por un lado, tenemos las variables con ámbito privado, las cuales funcionan de la misma manera que en los lenguajes convencionales, es decir, solo es accesible desde

dentro del contrato, y si queremos poder modificarlas o acceder a ellas, tenemos que definir funciones que actúen de *getters* o de *setters*.

Pero por el otro lado, tenemos las variables con ámbito público, las cuales podrá ser accedidas por cualquiera, ya sea una consulta al contrato por parte de un usuario, como por otro contrato, pero el concepto que hay que tener en cuenta es que cualquier usuario de la *blockchain* podrá consultar dichos valores.

Solidity define una función *get* implícita para todas propiedades que se hayan declarado públicas, por supuesto se pueden sobrescribir para hacer operaciones sobre el valor devuelto, pero por defecto será una función *get* normal.

El contrato cuenta con las siguientes variables:

- **IsAuthenticated**: Variable pública, inicializada por defecto a falso.
- **ExpirationTime**: Variable pública en la que se almacenará hasta cuando tiene vigor los valores de dicho contrato.
- **Owner**: Variable privada donde se almacenará la dirección pública del dueño del contrato. Para este proyecto es la variable más importante como comprobaremos a continuación.
- **Customer**: Variable privada para almacenar la dirección pública del usuario al que pertenece este *smart contract.*

Antes de proceder con el constructor del contrato, se define la función más importante de este *smart contract*, se va a crear una función de tipo *modifier*.

El concepto que hay sobre *modifier* es el de un decorador de funciones, el cual añade funcionalidades o restricciones a las funciones que decora [32] [31].

Para poder crear una función de tipo *modifier* deberemos de definir una función y sustituir la palabra reservada *function* con la palabra reservada *modifier*. Dentro de estas funciones deberemos de crear las funcionalidades extra o restricciones la cuales queremos añadir a otras funciones. Para ello, se crearán estas funcionalidades y se indicará mediante el símbolo "_" donde queremos que se ejecute la función a la cual *decora*, es decir en qué punto deseamos poner la función, puede ser antes o después de las funcionalidades añadidas.

Para que el concepto quede más claro, es una función que ejecuta funciones añadido funcionalidades extra o restricciones, sobre la función a ejecutar.

En este caso, solo vamos a definir una función *modifier* y además nuestra función definirá una restricción sobre las funciones a las que decore. Se añadirá una restricción, la cual antes de ejecutar la función dada, comprobará que la persona que está llamando a dicha función sea el dueño o creador del *smart contract*, de no ser así, la ejecución de la función acaba, se lanza un mensaje de error "*Caller is nor owner*" y no se prosigue con la ejecución de la función a la que decora.

De esta forma, nos asegurarnos, que en las funciones en las cuales le indiquemos este *modifier* solo se pueden ejecutar si son llamadas por el dueño del *smart contract*.

Una vez definida esta función, podemos comenzar con definición del constructor del contrato. En este constructor, utilizaremos la peculiaridad de las variables en *Solidity* al definir las variables como privadas estas solo pueden ser accedidas y modificadas desde el propio contrato, es decir, no ocultan información al exterior, ya cualquier usuario de la *blockchain* contará con una copia de dicha variable, pero nadie podrá modificar su valor.

Dicho esto, *Solidity* no define para las variables privadas ninguna función de tipo *get* o *set* de manera implícita, y en este caso nosotros no definiremos ninguna función *set* para estas variables, por lo que, una vez asignado el valor, no se podrá modificar de ninguna manera.

Para poder asignar valor a estas variables, se utilizará el constructor del contrato. El constructor del contrato, solo se ejecutará una vez cuando se despliegue el *smart contract* en la *blockchain* y nunca más se volverá a ejecutar. Estas particularidades, nos permiten tener variables con datos inmutables en el tiempo, y las cuales solo se asignará una vez.

Por todo ello, utilizaremos el constructor del contrato para asignar el valor a las variables privadas *owner* y *customer*. De esta forma la dirección del dueño o creador del contrato, así como la dirección pública del usuario, no podrán cambiar de valor nunca. Además, gracias a los principios de la *red blockchain*, nadie podría cambiar el valor aun teniendo todos los usuarios una copia de este *smart contract*.

Además de todo lo que ya hemos visto, debemos de saber, que, a este constructor, solo se le enviará la dirección del usuario. La dirección del propietario del *smart contract* se obtendrá del contexto de quién está desplegando ese *smart contract* en la *blockchain*, de

esta manera nos aseguramos, que el propietario sea siempre el cliente que esté creando la transacción con el *smart contract* dentro del bloque de la cadena.

Una vez tengamos el *smart contract* desplegado en la red de la *blockchain*, tendremos a nuestra disposición tres funciones, que utilizarán los clientes que interactúen con el contrato. Estas tres funciones serán de tipo *view* y públicas.

Los modificadores o decoradores de funciones que podemos encontrar en *Solidity*, además del decorador *modifier*, son las palabras clave *view* o *pure*. El decorador *view* indica que la función va a ser de tipo *solo lectura*. Es decir, que la función se ejecutará sin modificar o escribir ningún tipo de dato.

Por otro lado, las funciones que tienen el decorador *pure* indican que el resultado que va a devolver va a ser solo dependiente de los parámetros de entrada de dicha función y de ningún otro valor externo. Es decir, que no hace un tratamiento de los datos contenidos en el contrato, por ejemplo, una función que recibe dos números y devuelve la suma de estos.

La primera función se llama *GetCustomer* esta función tendrá el decorador *view*. De esta manera, la función devolverá la dirección del usuario para el cual fue desplegado el *smart contract* que se está ejecutando, pero solo el propietario del *smart contract* podrá consultarlo. Esta función ha sido definida para que todos los clientes del servicio pudieran utilizar la función para comprobaciones de seguridad, como veremos más adelante en el desarrollo de la API.

La segunda función se llama *SetAuthentication*. Esta función recibe como parámetros un *booleano*, el cual se utilizará para indicar si la autentificación es *verdadero* o *falso*, y un numero de gran tamaño para el tiempo de expiración, en este caso se trata de un número, ya que el tiempo de expiración estará expresado en *tiempo Unix* (es un sistema para la descripción de instantes de tiempo el cual se define como la cantidad de segundos transcurridos desde la medianoche UTC del 1 de enero de 1970) [33].

Esta segunda función, contendrá el decorador de funciones *isOwner*, ya que solo el propietario del contrato podrá modificar los valores que, de la autentificación del usuario, así como futuras características, roles o funciones.

Por último, la última función que podemos encontrar en el contrato se llama *GetOwner*. Esta función solo tendrá el decorador *view* ya que cualquiera podrá llamar a

esta función, para averiguar quién es el dueño o creador del *smart contract* que se está ejecutando. Esta función será utilizada por los clientes para comprobaciones dentro del ámbito *single sign on* como veremos más adelante en el apartado del desarrollo de la API.

5.3.2 Compilación del *Smart Contract* y ABI

Antes de poder empezar a trabajar en el desarrollo de la API, debemos de generar el ABI (Application Binary Interface o Interfaz Binaria de Aplicación) del contrato que hemos generado, todo ello para poder interactuar con el contrato, ya que debemos de compilar el ABI del *smart contract* en un formato en cual podamos importarlo y utilizarlo en nuestra API, la cual recordemos está desarrollada con el lenguaje de programación *Go*.

Para poder llevar a cabo este proceso, deberemos de instalar lo primero el compilador de *Solidity* en nuestro ordenador. Hay diferentes formas de tener el compilador en el ordenador, dependiendo del tipo de sistema operativo que se utilice. En el caso de Windows, podemos utilizar *Docker* para virtualizar una imagen ya preparada con todo lo necesario. En mi caso tras varias pruebas se va a utilizar el sistema operativo *Ubuntu* para llevar a cabo el proceso de compilación. En el sistema operativo *Ubuntu* podemos encontrar el compilador como un paquete de *snapcraft* (sistema de gestión de paquetes diseñado por Canonical) [34].

Otra de las herramientas que debemos de tener instalada en la máquina encargada de la compilación del *smart contract*, será la herramienta llamada *Abigen*, disponible dentro del paquete de la implementación de *Ethereum para Go*, el cual recordemos que es una implementación oficial del protocolo de *Ethereum.*

Para poder llevar a cabo el proceso de compilación del *smart contract* deberemos de tener instalado, en la máquina en la cual se llevará a cabo el proceso de compilación, *Go*, además todo lo nombrado anteriormente.

Una vez tengamos el entorno preparado, mediante línea de comandos, se realizará la compilación indicando el fichero *abi*, aunque este fichero se puede omitir si se añade la herramienta al comando de compilación, el nombre del paquete que se va a crear, el cual recordemos también será el *namespace* del mismo, en este caso el paquete se llamará *contract*, y por último, el *smart contract* que queremos compilar, en formato *.sol* (extensión de ficheros creados con el lenguaje de programación *Solidity*).

Una vez terminado el proceso de compilación, tendremos un paquete para el lenguaje de programación *Go*, mediante el cual podremos interactuar con nuestro *smart contract* gracias al protocolo de *Ethereum*.

5.3.3 Desarrollo de la API

Durante este apartado, se van a explicar las diferentes fases del desarrollo de la API, ya que estará compuesta de diferentes paquetes de *Go*, cada cual destinado con un propósito diferente.

Lo primero que veremos en este apartado del desarrollo de la API, será los nuevos paquetes generados para *Go*, con la intención de ver cómo se usan y se utilizan para interactuar con el *smart contract*.

Una vez visto cómo se interactúa con los contratos y la *red blockchain*, se explicarán los paquetes que conformaban el proyecto sobre el que se basa, para poder ver las mejoras o cambios introducidos. Muchos de los paquetes que conformaban las API han tenido que reajustarlos para poder funcionar bajo el nuevo modelo de gestión de identidades basadas en *blockchain*.

5.3.3.1 Servicio para interactuar con el smart contract

Dentro de la API, podremos encontrar un paquete que proporciona servicios al flujo de trabajo de la API. Dentro de este paquete podremos encontrar varios servicios que veremos en los siguientes apartados. Los servicios que conforman al paquete son:

- Servicio para la autentificación: en este servicio, se llevan a cabo los procesos de autentificación de los usuarios.
- Servicio para el registro de usuarios: en este servicio se llevan a cabo los procesos necesarios para el registro de los usuarios.
- Servicio para generar una sal criptográfica: este servicio se utilizará a lo largo de toda la aplicación siempre que se utiliza una función *PBKDF* (función de derivación de claves), se generará una sal criptográfica, la cual será una secuencia de bits aleatorios [35].
- Servicio para interactuar con el *smart contract*: este servicio es nuevo y se utiliza para interactuar con el contrato desde cualquier parte de la API,

haciendo uso del paquete *contract* generado en el apartado de compilación del contrato con nuestra máquina de compilación.

Uno de los servicios más importantes de todo el desarrollo es el servicio para interactuar con el *smart contract*. A continuación, se explica cómo funciona dicho servicio.

Dentro del servicio, la primera función que encontramos se utilizará cada vez que interactuemos con el contrato. Esta función se llama *getClient*, y en ella se configura el cliente. La configuración del cliente de *Ethereum* es fundamental para interactuar con la *red blockchain*. Para configurar el cliente se ha de proporcionar la URL del proveedor de la *blockchain*. En nuestro caso, durante el desarrollo se utilizará el software *Ganache* como proveedor de la *red blockchain*. Si la conexión con la *red blockchain* se realiza correctamente entonces se devolverá dicha conexión para poder comenzar a utilizar la red.

Una vez se ha obtenido la conexión con el proveedor de la *red blockchain*, lo siguiente que se debe de configurar son las propiedades de las transacciones, así como configurar la cuenta del dueño de ese *smart contract*, en este caso, la cuenta pertenecerá al cliente el cual está utilizando el servicio.

Las cuentas en *Ethereum* son o bien direcciones de monederos de criptomonedas o direcciones de un s*mart contract*, y son utilizadas para el envío y recepción de *ether* (criptomoneda descentralizada de *Ethereum*, que se utiliza también para la ejecución de contratos [3] [11]), estas direcciones son únicas y se obtienen a través de la derivación de una clave privada. Se explicarán con más detalle las cuentas de *Ethereum*, en el apartado del registro del usuario.

Continuando con la configuración, encontraremos una función, la cual se llama *getTransactOptions*. En esta función, lo primero que se hace es cargar la clave privada de la dirección del cliente. Esta se obtiene de un fichero de configuración del servicio. Una vez obtenida la clave privada, podremos derivar de esta la clave pública.

Gracias a la clave pública y a la clave privada, podemos obtener la dirección pública del cliente. Con esta dirección podemos obtener tanto el *nonce* (número único para identificar la transacción) como el *precio del gas* (el *gas* recordemos que es el coste computacional que requiere la transacción que vamos a realizar) este número será una aproximación de gas necesario a utilizar para que la transacción se puede llevar a cabo

correctamente, este gas se transformará a valor de *ether* durante el proceso de ejecución del contrato.

A todos los parámetros anteriores se añade un límite de *gas* el cual no queremos superar en caso de que la transacción por lo que sea tarde más en ejecutarse, en nuestro caso ese límite será de trescientos mil. Con todos estos parámetros vistos, ya tendríamos las opciones de las transiciones configuradas. Esta función y la función de conectar al proveedor han de realizarse siempre antes de proseguir con la transacción.

Una vez que tenemos todas estas características configuradas, ya podemos hacer uso del paquete *contracts* anteriormente generado para publicar un contrato en la *red blockchain*. Es por ello por lo que se ha creado una función llamada *DeployContract* la cual usará una función del paquete *contracts*, a la cual pasaremos los parámetros configurados en los párrafos de arriba, así como la dirección del usuario al cual pertenecerá el *smart contract*, recordemos que este era el parámetro de entrada para el constructor de contrato.

Sí todo se realiza correctamente se nos devolverá la dirección del contrato desplegado y una instancia del contrato, por si queremos interactuar con este directamente. Con este proceso habremos desplegado una instancia del contrato que hemos definido, para un usuario dado, en la red de Ethereum.

Al igual que se ha diseñado una función para desplegar un contrato en la red, también se ha diseñado otra, para poder cargar una instancia del contrato que esté desplegado en la red. Esta función se llamará *LoadContract* y en ella se utilizará de nuevo el paquete *contracts* el cual contiene un método para poder crear una instancia de un contrato cargándolo desde su dirección dentro de la red, por lo que a esta función se le pasará como parámetro, la dirección del contrato del cual se quiere crear una instancia. Si todo el proceso de creación de la instancia se ha realizado correctamente, se devolverá para poder interactuar con él.

Haciendo uso de esta instancia del contrato, que se ha cargado con la función anterior, se generan métodos para abstraer todavía más la recuperación y asignación de datos del contrato. Por ello, se realizan funciones a las cuales, pasándoles la instancia del contrato y el valor, estos se recuperan o actualizan en el *smart contract*:

- **GetAuthentication**: para recuperar el valor del campo *IsAuthenticated* del contrato.
- **SetAuthentication**: para cambiar el valor del campo *IsAuthenticated* del contrato.
- **GetExpirationTime**; para recuperar el valor de ExpirationTime del contrato.

Además de la función creada para recuperar la fecha de expiración presente en el contrato, también se ha generado una función, la cual es la encargada de comprobar que esa fecha de expiración recuperada del contrato no ha excedido la fecha actual, de ser así, esta función devolverá *falso* y no se permitirá seguir adelante con ningún tipo de comprobación o transacción.

Se han de comprobar más parámetros del contrato desplegado en la red para saber si podemos fiarnos de dicha instancia del contrato durante el proceso de *single sign on*. Es por ello, por lo que se ha generado una función llamada *CheckTrustClients*. En esta función se llevará a cabo la comprobación del dueño del contrato que se le pase como parámetro, de forma que, si el dueño el contrato no pertenece a nuestra lista de clientes de confianza, esta función devolverá *falso* y no permitirá continuar con ninguna operación de autentificación ni transacción a la red.

Estas comprobaciones del contrato se realizarán dentro de una función llamada *CheckContract*. La cual combinará las comprobaciones anteriores junto con el hecho de saber si el campo del contrato *IsAuthenticated* tiene un valor verdadero.

Con todas estas funciones y configuraciones, se ha creado un servicio el cual se encarga de interactuar con el contrato por nosotros, abstrayendo la complejidad a unas llamadas sencillas con las que poder usar el contrato en cualquier punto del flujo de la API.

5.3.3.2 Servicio para el registro de los usuarios

Este servicio es utilizado para interactuar con la base de datos tradicional de la que hace uso el servicio. En esta base de datos se almacenan tanto los usuarios del servicio, así como los clientes en los que queremos confiar.

En el caso del almacenamiento de los clientes en los cuales confiamos, en este caso se han almacenado en la base de datos, la dirección pública del cliente en la *red blockchain* que estamos utilizando y otro campo para indicar si se confía o no, de manera que podemos bloquear todas las transacciones con una simple actualización en la base de datos. En el apartado de mejoras futuras, se explicará las mejoras y la dirección que se tomará con respecto a la configuración de la lista de clientes en los cuales se confía.

Para el caso del registro de un usuario, se partirá del método del proyecto base, y se introducirán nuevas características y mecánicas para este nuevo.

Lo primero que se realizará en el método encargado de realizar el registro de los usuarios, será la deserialización del cuerpo de la petición realizada a la API, el cual contendrá todos los datos del usuario a registrar.

Una vez se han obtenido todos los datos, deberemos de realizar las comprobaciones correspondientes, como por ejemplo si el email proporcionado es un email valido, o si el usuario que se intenta registrar ha proporcionado una contraseña fuerte, o por el contrario ha delegado la creación de la contraseña en el servicio. Si todas estas comprobaciones son realizadas correctamente, se procederá a establecer la conexión a la base de datos.

En la base de datos, un usuario se caracteriza por tener las siguientes propiedades: email, contraseña, una sal criptográfica y la dirección pública de usuario de la *red blockchain*, cuando nos referimos a dirección pública, es la cuenta de un usuario de *Ethereum.*

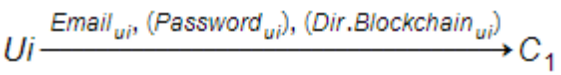

Figura 3 Envío de datos del usuario al cliente

Para poder guardar esta dirección del usuario de *Ethereum* el usuario debe de proporcionar esa dirección. ¿Pero qué ocurre si el usuario no dispone de un monedero de criptomonedas o un monedero de direcciones de *Ethereum*? Bien, en este caso, también se puede delegar la creación de una dirección para el usuario dentro de la red de *Ethereum*. Para este cometido, dentro del paquete *core* se han creado funciones que se ocupan de crear estas direcciones.

Una vez conectado a la base de datos y con toda la información del usuario validada, procedemos a realizar *hashes* de dichos datos, para almacenarlos en la base de datos. Lo primero que haremos, será utilizar la función de derivación de claves Argon2.

Utilizaremos para ello el paquete "crypto/argon2" y la función "IDKey", contenida en el paquete, la cual implementa la función Argon2id. Para poder utilizar esta función de derivación de claves, deberemos de indicarle parámetros para poder crear el resumen utilizando una serie de recursos. Los parámetros han de ser elegidos para conseguir un correcto funcionamiento junto a un buen rendimiento para esta función de Argon2id [36].

Los parámetros que he definido para la función son los siguientes:

• Numero de bytes mínimos para la sal criptográfica: 32.

• Número de iteraciones sobre la memoria: 1.

• Memoria usada: 64 * 1024

• Hilos utilizados: 4.

• Longitud mínima de clave: 32 bytes

En este caso para la función de derivación de claves se generará una sal criptográfica única para cada usuario, mediante el servicio de generación de sal, y junto a esa sal se utilizará la función de derivación de claves de Argon2id con la contraseña. También se realizará el *hash* del email, pero con una sal criptográfica diferente a la utilizada para la contraseña.

$$r(Email_{ui}) = h^{Argon2id}(Email_{ui})$$
$$r(Password_{ui}) = h^{Argon2id}(Password_{ui})$$

Figura 4 Creación de un hash para el email y la contraseña del usuario

Una vez hemos aplicado la función de derivación de claves, se codificará el resultado en *base64* y se almacenará en la base de datos.

$$C_1 \xrightarrow{r(Email_{ui}),\ r(Password_{ui}),\ Sal,\ Dir.Blockchain_{ui}} BD$$

Figura 5 Registro del usuario en la base de datos

En caso de que todo el proceso relacionado con la base de datos haya salido correctamente, procederemos a enviar la respuesta al usuario. En caso de que se haya delegado en el servicio la creación de contraseña o la creación de la cuanta de *Ethereum*, estos datos también serán devueltos en la respuesta.

5.3.3.3 Creación de direcciones

Como ya se comentó anteriormente, las cuentas de *Ethereum* son direcciones de cateras, las cuales están formadas por un par de claves *ECSDA* que, a su vez, se almacenan en una cartera de direcciones, monedero de criptomonedas o bien direcciones de un *smart contract*, al final todos estos aspectos identifican a alguien en la *red blockchain*.

Las cuentas pueden ser utilizadas para transferir *Ether* o enviar mensajes y con ello interactuar con *smart contracts*. Las cuentas son únicas y dependen de una clave privada.

En nuestro caso, antes de poder trabajar con una cuenta de *Ethereum* deberemos de transformarla, es decir pasar la cuenta de formato cadena a formato *address* de la implementación de *Ethereum en Go*.

Llegados a este punto, podemos ver que la cartera de direcciones al final no es más que una manera de almacenar una clave privada y una pública, las cuales se utilizaran para firmar transacciones, y utilizando la clave pública para generar dirección única y pública también que identifique al usuario dentro de la *red blockchain*.

Hay tres maneras de crear carteras con las claves: carteras o monederos normales, ficheros *keystores* y carteras o monederos HD.

Para generar las claves y con ello las direcciones, en de los monederos de direcciones normales deberemos de generar una clave privada, la cual se genera con la función especializada en ello de la implementación de Ethereum para *Go*.

Tras haber obtenido esta clave privada, esta deberá de ser tratada como una contraseña y nunca deberá de ser compartida por nadie, ya que con esta clave será con la que se firmaran todas transacciones que realicemos en la *red blockchain*. Si en algún momento perdieras tu clave privada perderíamos todo lo que hace referencia a la cuenta,

como por ejemplo las criptomonedas asociadas a esta, ya que la cuenta no se podría recuperar.

Siguiendo con la clave pública, esta es derivada de la clave privada siempre y cuando haga falta. Gracias a esta clave pública, podremos genera una dirección pública del usuario, la cual será un identificativo de la cuenta.

Esta dirección es un *hash* realizado con *KECCAK-256*, la cual es una función similar a *SHA-3*, ya que fue desarrollado por el mismo equipo, solo que se diferencian en el relleno o *padding* que se realiza en la función. *Ethereum* utiliza esta función de resumen en todos los *hashes* que se realizan [37].

Por lo que para poder identificarte como usuario al que pertenece esa cuenta, deberás tener la clave privada, ya que solo el propietario o el que la tenga en su posesión, será capaz de generar la clave pública y la dirección, las cueles derivan de esta clave privada.

De esta manera, ya tendríamos generadas las claves necesarias para la cartera de direcciones. Como se puede observar, no es lo más aconsejable dejar que el servicio te generé los pares de claves, pero sí que puede llegar a ser de utilidad según los fines con los que nos estemos registrando en el servicio.

Es por esto último por lo que se desarrolló los archivos *Keystore*. Un archivo *keystore* almacena tu clave privada cifrada con una contraseña. Por lo que si abrimos el archivo solo veremos la clave privada codificada y cifrada con la contraseña que hayamos proporcionado.

Durante la generación de la generación de una clave privada en un archivo *Keystore* deberemos de proporcionar una contraseña con la que se realizará el cifrado. Lo interesante de este método, es que podemos cambiar la contraseña y al hacerlo se generara un fichero nuevo, teniendo en cuenta que la integridad de tu fichero *Keystore* será proporcional a como de fuerte sea tu contraseña.

Por último, tenemos las *HD Wallets* o *Hierarchical Deterministic Wallets*, estas carteras se caracterizan por poder generar direcciones, basadas en una semilla. Las semillas pueden ser de doce, dieciocho o veinticuatro palabras clave [38]. Recordar la semilla es difícil debido a que son caracteres alfanuméricos aleatorios, por lo que se utilizan mecanismos mnemotécnicos para convertir la semilla a palabras entendibles para el ser

humano. Las semillas están basadas en un protocolo de mejora de *Bitcoin* conocido como *BIP32* [39].

Con este tipo de monederos, se resuelve la problemática de tener que generar direcciones de manera manual y la creación de copias de seguridad. Si se pierde el monedero, se puede llegar a recuperar la clave privada, gracias a semilla, por lo que con este tipo de carteras también resolvemos el problema de la perdida de la clave privada.

Una vez vistas las tres formas de generar cuentas de usuario de Ethereum, de momento el servicio va a generar carteras de direcciones normales y ficheros *Keystore*. Esto es debido a la complejidad de implementación de los monederos HD, la cual nos llevaría mucho tiempo y escapa a la finalidad del trabajo, por otro lado, se propondrán mejoras futuras para la creación de la cuenta de los usuarios.

Volviendo a la implementación, dentro del paquete *core* se definirán las funciones *generatePrivateKey* y *generatePublicKey*, las cuales como su propio nombre indica, se utilizan para crear la clave privada y pública respectivamente. Estas funciones a su vez son utilizadas por la función *GeneratePublicAddress* la cual utiliza la clave pública y la privada para crear la dirección pública del usuario.

Con todas estas funciones ya podremos generar el par de claves y la dirección necesarias para que los usuarios puedan usar el servicio. Por otro lado, se han creado dos funciones para la creación y la importación de ficheros *Keystore*.

5.3.3.4 Servicio de autenticación de usuarios

Dentro del servicio implementado para la autentificación de usuarios, se definen varias funciones con las cuales podremos llevar a cabo esta tarea.

La primera función que encontraremos dentro de este servicio será *LoginService*, la cual es la función más importante que se verá en este apartado, ya que es en esta, en la que se realiza toda la lógica de la autenticación de usuarios.

Esta función recibirá un objeto de tipo *usuario* ya que como veremos más adelante en otros apartados, antes de llegar a esta función del servicio, se realizarán más comprobaciones.

Se considera que, si el flujo de la API ha llegado hasta esta función del servicio, el usuario recibido por paramento es un usuario valido, por lo que, en primer lugar, se realizará la conexión a la base de datos.

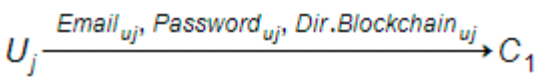

Figura 6 Envío de datos para la autenticación

Una vez hayamos establecido la conexión a la base de datos, se ha de recuperar el usuario de esta, pero recordemos que todos los datos almacenados, han sido transformados por la función de derivación de claves Argon2id. Por lo que, para poder realizar búsquedas en la base de datos basadas en el email del usuario, deberemos de realizar un *hash* con la misma función de derivación de claves y la misma sal.

Este proceso es posible ya que se utiliza la misma sal criptográfica para todos los emails y solo los emails, es decir para cada usuario, se crea siempre una sal criptográfica única y aleatoria que se utilizará para el *hash* de la contraseña.

Cuando se tiene el hash del email, se recuperarán los datos del usuario de la base de datos para ese email dado. Es en este momento en el flujo del servicio de autentificación de usuarios, cuando se comprueba el segundo factor de autenticación. El segundo factor de autenticación biométrico en el desarrollo de este proyecto ha sido la huella dactilar. Cuando se recupera el usuario de la base de datos también se recupera el estado de los segundos factores de autentificación, estos estados pueden tener valores de *verdadero* o *falso*.

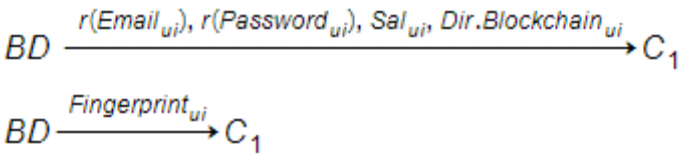

Figura 7 Recuperación de los datos para un usuario dado

En caso de que la sesión haya caducado, siempre se pedirá el segundo factor de autentificación y en caso de que la sesión siga activa, esta autentificación mediante

segundo factor no se llevará a cabo. La vida útil de una sesión para el servicio es por defecto de una hora.

Siguiendo con el flujo, si es necesario que el usuario se autentifique con el segundo factor de autentificación, se pausará el flujo de la llamada, durante dos minutos. Durante este tiempo el servicio intentará recuperar valores correctos para alguno de los segundos factores de autentificación. En caso de que el usuario introduzca valores válidos para alguno de los segundos factores disponibles, el servicio continuará con la autentificación del usuario. Pero en caso de que no se introduzcan valores válidos durante este periodo de tiempo, el flujo de la autentificación del usuario se aborta y se lanza un mensaje de error, para indicar al usuario que no ha sido posible autentificarle y que debe de volver a comenzar el procedimiento.

$$U_j \xrightarrow{Fingerprint_{uj}} C_1$$
$$Fingerprint_{uj} = Fingerprint_{ui}$$

Figura 8 Comparación de los valores para los segundos factores de autenticación

Si toda la autentificación del segundo factor de autentificación ha funcionado, se continuará con la comprobación de la contraseña que el usuario ha introducido. Para poder realizar la comparación de las contraseñas, primero deberemos de aplicar la función de Argon2id a la contraseña se ha recibido en la petición, para poder compararla con la que ya teníamos almacenada.

La función de derivación de claves se aplicará con los mismos parámetros que ya vimos en el apartado del registro del usuario y se utilizará la sal criptográfica única para dicho usuario. Como resultado obtendremos un array de bytes, el cual se comparará byte a bytes con el que tenemos almacenado en la base de datos.

Si las contraseñas coinciden, se procederá a comparar si la dirección pública del usuario dentro de la red *Ethereum* para ver si coincide con la que tenemos almacenada en la base de datos, la cual también se habrá enviado en la petición de autentificación. Si la dirección es igual byte a byte con la que tenemos almacena en la base de datos, significará que el usuario que intenta autentificarse es legítimo, por lo que procederemos a generar un token JWT y para enviar en el cuerpo de la respuesta al usuario.

$$r(Password_{ui}) = h^{Argon2id}(Password_{uj} + Sal_{ui})$$
$$Dir.Blockchain_{uj} = Dir.Blockchain_{ui}$$

Figura 9 Comparación de contraseñas y direcciones públicas de blockchain

Para la generación del token, se utilizará una función creada en el mismo paquete que este servicio de autentificación, llamada *TokenResponse*. En este método se llamará a la función encargada de generar el token JWT, la cual se verá más en detalle en los siguientes apartados, pero, una de las cosas que debemos de saber es que, en el interior del token, se almacenará el email del usuario.

Junto con la generación del token se obtiene la fecha de expiración del mismo, que corresponderá con la fecha de validez de la sesión para dicho usuario. Esa fecha de expiración de la sesión será la que se grabará en el *smart contract* del usuario, por lo que en este punto se conseguirá una instancia de contrato del usuario utilizando su dirección pública de la *red blockchain*.

Con la instancia del usuario, se utilizará el servicio creado para el manejo de los contratos y se realizará una transacción grabando en el contrato la fecha de expiración de la sesión.

Si todo el procedimiento de autentificación del usuario se ha realizado correctamente, se le devolverá una respuesta con el token el cual se utilizará en el servicio del cliente para poder interactuar con él y además se le devolverá la fecha de expiración del token, para que el cliente administre el refresco de dicho token JWT.

Dentro de este servicio, también podemos encontrar funciones auxiliares para poder manejar todo este flujo de autentificación de usuarios. Entre estas funciones encontramos *CheckMultiFactor*, utilizada para la comprobación del estado de los todos los segundos factores de autenticación de los usuarios. Junto a esta función también encontramos la función *InsertMultiService* la cual se utiliza para realizar las inserciones de los valores de los segundos factores de autentificación en la base de datos.

Lo único que nos queda por ver, es una función llamada *RefreshTokenService*, utilizada para el manejo del refresco de la fecha de expiración del token de los usuarios.

5.3.3.5 Validación del token

El token del usuario deberá de utilizarse en todas las peticiones con las que se vaya a interactuar con el servicio. Al igual que esta validación, se deberán de realizar otras muchas más conforme vaya avanzado el servicio. Para no repetir funcionalidades a lo largo de la API, se va a diseñar un *middleware* el cual se utilizará en el control de todas las llamadas que se reciban.

Un *middleware*, también es llamado lógica de intercambio de información entre aplicaciones, normalmente se habla de *middleware* como software que permite a una aplicación comunicarse con otras aplicaciones, paquetes de programas, redes, maquinas (refiriéndonos a máquina como ordenador) o incluso con el sistema operativo. También es conocido por ser software que se utiliza para conectar sistemas distribuidos [2] [22].

En este caso, se le ha llamado *middleware* a un paquete desarrollado para el proyecto, el cual permitirá el paso o no de las peticiones a la lógica de negocio de la API, es decir, este paquete actuar como una barrera ante las peticiones entrantes. Aunque no cumple con la definición propia de lo que es un *middleware*, pero su nombre nos deja ver cuál será el propósito del paquete, ya que permitirá la comunicación entre software, como propiamente se ha explicado.

De esta manera se conseguirá una abstracción de la comunicación, ya que todas las llamadas definidas en los controladores de peticiones a las cuales se les añada este *middleware* tendrán que pasar por las verificaciones que se definan.

Para la generación de tokens de tipo JSON Web Tokens sobre una implementación en *Go*, se utilizará el paquete *dgrijalva/jwt-go* [40], este paquete nos proporcionará la implementación que necesitamos de estos tokens para *Go*.

Este paquete de *middleware* encontraremos dos funciones importantes, la primera de ellas se llama *TokenAuthentication*. En esta función, lo primero que realizamos es extraer el JWT de la petición, que estará en la cabecera *Authorization*. Una vez tengamos el token, deberemos de comprobar que ese token es legítimo y que ha sido creado en nuestro servicio, para ello se comprobará que el token JWT ha sido firmado con nuestras claves *ECSDA*, en los siguientes apartados se describirá como se genera dicho token. Si el token no ha sido firmado por nosotros, se emitirá un mensaje de error que se registrará en el log de la API y no dejará pasar la petición al flujo del servicio.

Si el token obtenido de la cabecera si ha sido formado y firmado con nuestras calves *ECSDA*, entonces se pasa a verificar la integridad del token, es decir, comprobar que contiene todos los atributos y que este se formó correctamente. Si el token es válido, solo nos quedará realizar la última comprobación, la cual será comprobar si el tiempo de expiración contenido en el token ha sido excedido.

Para poder comprobar los atributos contenidos en formato JSON en el token, deberemos de extraer los *claims* token, como por ejemplo la fecha de expiración o algunos atributos más, como la dirección del contrato del usuario. Si todo el proceso ha ido correctamente la función deja pasar la petición a la siguiente función o controlador y termina.

En caso de que el token haya expirado o no sea válido en cuanto al formato, ocurren tres cosas, la primera es que se anulan todos los posibles valores de todos los posibles métodos de segundo factor de autenticación que tengamos en la base de datos para el usuario del token.

Lo segundo es que se realizará será la invalidación de atributos, roles y características dentro del *smart contract*. Para ello en este punto, se cargará una instancia del contrato y se realizarán los procesos para todas las invalidaciones, incluido el tiempo de expiración, el cual será la hora actual al proceso. Por lo que toda la identidad y características del usuario quedan invalidas en el *smart contract*.

Durante todo este proceso ocurre lo mismo que cuando haces *logout* en el servicio. Por último, se escribe en la cabecera de la respuesta el código de estado 401 *Unauthorized* y no se continúa procesando la petición.

5.3.3.6 Single Sign On en el servicio de autenticación

Para poder llevar a cabo esta funcionalidad con el servicio, se utilizará la segunda función importante que podemos encontrar en el paquete *middleware*, la cual se llama *blockchainAuthentication*.

Esta función será utilizada cuando el usuario intente realizar la autenticación con otro cliente diferente, pero el cual utiliza el mismo sistema de autenticación y en el cual se confía, ya que, aunque el cliente utilice este servicio de autenticación, no se podrá hacer *single sign on* si no se tiene confianza en el cliente que primero autentico al usuario.

Todo este proceso de autenticación comenzará por el usuario preguntando o demando un desafío, el cual deberá ser entregado por el cliente al usuario. Durante el desarrollo de este proyecto ya se ha explicado que se cuenta con funciones generadoras de token JWT y con funciones que comprueban dichos tokens. En este caso como ya tenemos disponible y funcional este mecanismo, el desafío que el cliente entregará al usuario, será un token JWT. Este hecho a su vez nos trae un veneficio y es que el usuario podrá verificar el token, debido a las propiedades propias de un token JWT, ya que dentro del token podemos encontrar entre sus múltiples partes, la firma del token, la cual es generada por un *hash* de la cabecera y el *payload* de este, firmado por el cliente que lo ha generado, el algoritmo utilizado y la clave pública o el secreto (según el algoritmo utilizado). De esta manera, si el usuario quiere verificar la firma, tan solo deberá de utilizar el algoritmo utilizado y los parámetros descritos para comprobar que la firma es auténtica y el token no ha sido modificado en el trascurso de la comunicación. Además, este token contendrá un tiempo de expiración de un minuto.

Una vez el usuario ha recibido el desafío y haya comprobado la firma del mismo, el usuario deberá de cifrar dicho desafío con su clave privada para fírmalo. Es en este punto en el que el usuario deberá comenzar el proceso de autenticación enviando al cliente el desafío firmado, junto a la dirección del contrato, en el cual el cliente ha sido autentica previamente.

$$C_1 \xrightarrow{Token} U_i$$
$$c_{T_{ui}} = E_{K_{PRIV_{ui}}}(Token)$$
$$U_i \xrightarrow{c_{Tui},\ Dir.SmartContract} C_1$$

Figura 10 Proceso de firma del desafío para el login single sign on

El cliente, cuando reciba la llamada de *login* junto con la dirección del contrato y el desafío firmado, lo primero que se intentará realizar será verificar la firma del usuario que intenta autenticarse.

Antes de explicar cómo funciona el proceso de verificación de firma, tenemos que hablar sobre las claves que utiliza *Ethereum*. Las claves utilizadas en *Ethereum* son claves basadas en curvas elípticas, en concreto son creadas con el algoritmo *ECSDA* (Elliptic Curve Digital Signature Algorithm), el cual es una modificación del algoritmo *DSA*

(*Digital Signature Algorithm*) para poder afrontar problemas de este último. *ECSDA* ha sido pensado para proporcionar la misma o más seguridad que el algoritmo *RSA* con claves más pequeñas [41] [42].

La firma con claves *ECSDA,* consienten el un algoritmo de firma [43] [42] el cual obtiene un mensaje a firmar junto con la clave privada y produce una firma que consiste en un par de numero enteros *{r,s}*. A su vez el algoritmo de comprobación de la firma recibe el mensaje firmado, es decir *{r,s}* y la clave pública con la que se ha comprobar la firma. De este modo el algoritmo comprueba que la firma sea correcta.

En el lenguaje de programación *Solidity* y en el protocolo de *Ethereum en Go*, así como en muchos otros lenguajes contamos con funciones con la posibilidad de comprobar la firma desconociendo la clave pública del usuario. Gracias a la característica de la firma con claves ECDSA, se puede descomponer la firma y encontrar cual es la dirección o clave públicas asociada a esa firma.

Esta función que nos permite saber la dirección de una firma se llama *ecrecover*. A esta función deberemos de pasarle tanto el mensaje firmado, como el mensaje sin firmar. Su funcionamiento en Solidity nos devolverá la dirección pública asociada a dicha firma [44] y dentro del paquete *crypto* disponible para la implementación del protocolo de *Ethereum en Go*, esta función nos devolverá la clave pública asociada a dicha firma. Haciendo uso de esta función, se realizará la comprobación de la firma del usuario.

En una primera aproximación, se ha diseñado de manera que la comprobación de la firma se realice en el cliente, el cual quiere autenticar al usuario. Pero debido a que el problema logarítmico de las curvas elípticas requiere más computación que la factorización de numero primos grandes y que es más seguro que se realice esta comprobación en *Solidity* se han propuesto soluciones de para poder pasar esta comprobación de firma a un *smart contract*, como se podrá leer en el apartado de líneas futuras.

Por lo que siguiendo con el flujo de autenticación *single sign on*, cuando el usuario envía el desafío firmado, también enviará la dirección del contrato en el cual cuenta con una autenticación, roles y características correctas y concretas.

Del contrato se recuperará también la dirección del usuario al que pertenece, el dueño del contrato, la fecha de expiración y todos los demás elementos que se necesitan para la comprobación, así como toda la información necesaria que se requiera del contrato.

Una vez hemos recuperado la información del contrato, lo primero que se comprobará, antes de nada, es si se confía en el cliente que desplego el *smart contract* que acabamos de consultar.

En caso de que sí se confíe en el cliente, se pasará a comprobar la firma con la clave pública asociada a la firma, así como la comprobación de que la dirección pública contenida en el contrato es la misma que se ha podido obtener derivada de la clave pública asociada a la firma. Si la firma y la dirección pública son correctas, deberemos de comprobar también la integridad del desafío, comprobando que efectivamente el token JWT emitido por nosotros es válido.

$$SmartContract \xrightarrow{Info_{uj},\ Dir.Blockchain_{uj},Owner_{uj}} C_1$$
$$K_{PUB_{uj}} = encover(c_{T_{ui}},\ Token)$$
$$Token = E_{K_{PUBuj}}(c_{T_{ui}})$$

Figura 11 Recuperación y comprobación de clave pública para el usuario dado

Con todas estás comprobaciones habremos comprobado que el usuario que está solicitando la autenticación, es legítimo y posé la clave privada a la que pertenece la dirección pública del *smart contract*.

Por último, se ha de comprobar que la fecha de expiración y todas las características del contrato son válidas. Si todo el proceso de comprobaciones se ha realizado correctamente, como se confía en la autenticación y las características creadas y validadas por un cliente tercero, para ese usuario, el sistema genera un token JWT de sesión y se le envía una respuesta al usuario con dicho token, el usuario habrá sido autenticado.

En caso de que alguna de las comprobaciones falle, no se permitirá el *login single sign on*, por lo que se redirigirá el flujo al servicio de autenticación, donde se realizará el *login* como se ha descrito en el apartado del servicio de autenticación.

5.3.3.7 Creación del token y contraseñas

Como hemos visto en los apartados anteriores, a la hora de crear direcciones públicas para los usuarios o a la hora de generar contraseñas, se estaba utilizando el

paquete del proyecto, llamado *core*. En este paquete encontraremos métodos ya descritos para la generación de direcciones, otros para la generación de contraseñas, el cual no se ha explicado, y finalmente uno para la generación de los tokens de tipo JWT.

En cuanto a la generación de contraseñas, hay que comentar que las contraseñas podrán ser generadas según los parámetros que indiquemos. Podemos seleccionar la longitud de la contraseña, si deseamos mayúsculas, minúsculas, números, símbolos o una combinación de ambos, con la limitación de que el tamaño mínimo de contraseñas será de ocho caracteres, esta restricción es debida a que una contraseña con menos caracteres se considera débil contra ataques de fuerza bruta. En un futuro no muy lejano, lo recomendable deberán de ser contraseñas de once caracteres.

Para las opciones de configuración de una contraseña, es decir para las mayúsculas, minúsculas y los símbolos, tendremos un mapa por cada una de estas opciones. Un tipo *mapa* o *map* en *Go*, es lo que se conoce en otros lenguajes de programación como diccionarios, es decir estructuras de clave/valor. En estos mapas, nuestra clave será un número, para poder usar el número pseudoaleatorio para acceder al mapa y el valor será de tipo *rune* (32-bits de valor entero) para representar un carácter *Unicode*, de esta manera podremos representar todos los símbolos de *Unicode*.

De este modo, usando el mapa junto a los números pseudoaleatorios, podemos generar caracteres del tipo que queremos de una manera pseudoaleatoria, que, al unirlos, nos dará la contraseña para el usuario. En el caso de que además queramos una contraseña con números, tan solo utilizaremos el generados de números pseudoaleatorios, para generar números comprendidos entre el 0 y el 9, ambos incluidos, de esta forma uniéndolo a los mapas anteriores, podemos introducir número de manera pseudoaleatoria a las contraseñas.

Si el usuario no indica que tipo de contraseña quiere, es decir no indica la confinación con la que quiere que se cree la contraseña, por defecto se creará una contraseña con minúsculas, mayúsculas y números, con un tamaño de ocho caracteres, siguiendo un poco los estándares de lo que hoy en día se considera una buena contraseña, por ejemplo, por defecto se crearía una contraseña parecida a esta: KirBy43v.

Por otro lado, como se ha mencionado, dentro del paquete *core* se crean también los tokens JWT. Durante el proceso de validación, ya vimos que una de las cosas que se comprueban del token, será la firma que este trae. Para poder firmar los JWT, utilizaremos nuestras propias claves pública/privada de *RSA*, por lo que, lo primero que hará el servidor

web de la API cuando arranque, será cargar dichas claves del directorio que nosotros le indiquemos, en memoria para poder crear los JWT.

Dichas claves serán almacenadas en un fichero, las cuales situaremos donde nosotros queramos, pero siempre indicando los medio o caminos necesarios, para poder acceder a ellas. Una vez hayamos cargado los bytes pertenecientes a las claves, utilizaremos el paquete *crypto/x509* [45] de la librería estándar de *Go*, para transformar los bytes a una estructura de claves públicas, siguiendo el estándar de *ITU* (Unión Internacional de Telecomunicaciones) [46].

Se cargan durante el arranque del servidor, debido a que esta tarea supondría bastante carga de trabajo si lo tuviéramos que hacer en cada llamada, de esta manera tendremos cargadas las claves en variables y accederemos mediante punteros.

Dentro de este paquete encontraremos una función llamada *GenerateToken*, la cual recibirá el email del usuario para poder introducirlo en el cuerpo del token.

Para generar un JWT, lo primero que debemos de hacer, es generar un token vacío, indicándole al método de firma que vamos a utilizar y cómo será el hash que se genera para la firma, en el caso de este proyecto se utilizará SHA-512.

Después se calculará el tiempo de expiración que le queramos dar al token, lo más aconsejable es lo que se ha desarrollado en este proyecto, que es un tiempo máximo de una hora de vida. Esta fecha de expiración tendrá un formato Unix, es decir, con formato UTC y en segundos.

Ya con la fecha de expiración calculada, se deben de crear los *claims*, que será en contenido que formaran el cuerpo del token. Los *claims* cuentan con una estructura en forma de clave/valor. En el caso de este proyecto y haciendo una primera aproximación hemos indicado tres *claims*, pero estos se pueden ir ampliando conforme el proyecto vaya creciendo en el caso de que nos haga falta.

Lo más aconsejable a la hora de generar las *claims*, es seguir la estructura estándar que hay definida para las claves. En una primera aproximación el token tendrá la siguientes *claims*:

• ***email***: como su propio nombre indica, contendrá el email del usuario.

• ***exp***: albergara el tiempo de vida del token en formato Unix.

• ***nbf***: el instante en el tiempo en el que se ha generado el token, también en formato Unix.

Una vez hayamos creado la estructura de los *claims*, solo nos queda realizar el proceso de firma del token utilizando nuestra clave privada de RSA, sobre el hash realizado del cuerpo y la cabecera del mismo token.

Una vez se ha terminado todo este proceso, se devolverá este token y la fecha de expiración, a la llamada que lo haya solicitado.

5.3.3.8 Controladores y llamadas a la API

Cuando hablamos de controladores en este proyecto, nos referimos a las funciones encargadas de recibir una llamada o petición desde un cliente al servidor donde está alojada la API.

Estas funciones son llamadas así, porque controlan el flujo que realiza la llamada una vez entran el servidor, decidiendo cual es el servicio que tiene que atender a dicha llamada. En estas funciones controladoras, es donde se indicarán los *middlewares* por los que deberá pasar la petición.

Por último, estás sunciones también se encargan del análisis y deserialización del cuerpo del mensaje, así como las encargadas de escribir las cabeceras y cuerpo de las respuestas para una petición dada.

Todas las rutas, se definirán en el paquete *routes* que podemos encontrar en el interior del proyecto. Las rutas son definidas mediante una URL y se agruparán en ficheros independientes según el propósito de las mismas. Además, las rutas también se versionarán, con el fin de que en un futuro se puedan realizar actualizaciones en la API sin romper las API que en ese momento esté en ejecución.

Debido a esta característica, las rutas tendrán un aspecto muy similar al siguiente: *http://IpServidor:443/v1/login*.

Al ser una API, tendremos una documentación de esta, la cual podrá ser encontrada como anexo a este documento.

6. Conclusiones

Partiendo del trabajo de final de grado *Desarrollo de un servicio de autentificación de factor múltiple* [2], se ha conseguido desarrollar un sistema de autenticación de usuarios con gestión de identidad basada en *blockchain*, consiguiendo así un sistema en cual se puede lograr un *login single sign on*.

Como este proyecto ha sido diseñado con una estructura que utiliza la *red blockchain* desde el principio, nos permite que el sistema evolucione junto con la tecnología *blockchain*, de modo que todas las nuevas funciones, así como nuevas soluciones de la misma serán integradas de manera automática en este proyecto.

Además, gracias a las versiones y continua evolución del lenguaje de programación *Solidity*, podemos contar con múltiples *smart contracts* en diferentes versiones. Lo que nos permite un correcto funcionamiento del sistema durante el desarrollo de nuevos contratos para la evolución del lenguaje.

Todo este desarrollo comenzó con el objetivo de ser un servicio que pudiera aproximar la identificación mediante identidades en *blockchain*, a las soluciones actuales. Con este desarrollo hemos podido ver que una evolución hacia sistemas de identidad soberana en la Web 3.0 es posible y que, tarde o temprano, se acabaran imponiendo.

Todo el servicio ha sido pensado para ser seguro y privado en todos los aspectos de autenticación de usuarios. Al igual que ha sido pensado para ser seguro en todos sus puntos, también ha sido pensado para poder obtener un gran rendimiento con pocos recursos, sin tener que perder seguridad en el camino.

El haber sido desarrollado con el lenguaje de programación *Go*, nos permite que este servicio se pueda integrar en cualquier sistema operativo o hardware. Otro punto para tener en cuenta es que, gracias al lenguaje, que cuenta con sistema de paquetes, podemos añadir funcionalidades nuevas de una manera fácil y sin interferir en flujo del servicio. Este proyecto tiene un gran potencial y puede ampliarse fácilmente, puesto que las tecnologías asociadas a la autenticación de usuarios evolucionan continuamente.

El servicio desarrollado junto con las mejoras nombradas en las líneas futuras previamente, conformarían un servicio realmente potente de autenticación de usuarios con *login single sign on*.

6.1 Problemas encontrados

En líneas generales el desarrollo de este proyecto no ha conllevado problemas graves que nos impidieran continuar con el desarrollo del mismo.

Uno de los mayores problemas al que me he tenido que enfrentar durante el desarrollo de este proyecto ha sido la documentación y datos sobre *blockchain*. A lo ancho y largo de Internet, podemos encontrar muchísima información sobre *blockchain* y las diferentes formas de aplicarlo a múltiples campos muy diferentes entre sí. Mucha de la información que podemos encontrar al respecto en Internet solo intentan vender *blockchain* para ser usado en casi cualquier situación, incluso en situaciones en las que no tiene sentido aplicar dicha tecnología ya que, en la mayoría de las situaciones, la tecnología actual o convencional tiene muchos más beneficios.

Blockchain es visto en muchas ocasiones, como la panacea a todos los problemas que existen con la tecnología actualmente e, incluso, como solución a problemas que escapan del ámbito de la tecnología. Por todo ello, se hace necesario un enfoque crítico con esta tecnología antes de aplicarla a situaciones o casos que se plantean.

Como el mundo de *blockchain* se mueve entorno a las criptomonedas, hay que tener mucho cuidado también con la información que leemos pues, en muchas ocasiones, pueden ser una estafa para el usuario que está buscando información. En otras ocasiones la información que podemos encontrar solo se queda con el lado de las criptomonedas y no muestran realmente el potencial que podría llegar a tener esta tecnología.

Hay que recurrir siempre a la documentación oficial de los proyectos de las *blockchain*s. En el caso de *Ethereum*, en la página oficial del proyecto podemos encontrar toda la información necesaria, tanto para la parte de criptomonedas y poder llegar a entender bien cómo funciona, así como la parte del desarrollo, tanto de la propia *blockchain* como el desarrollo de los *smart contracts*. Además, toda esta documentación está disponible en muchos idiomas y contamos con chats exclusivos de la comunidad para poder preguntar dudas al respecto, como por ejemplo como desarrollar una funcionalidad en *smart contracts* [4] [47].

Llegados a este punto, otro problema al que me he tenido que enfrentar ha sido la necesidad de adquirir los conocimientos necesarios sobre la propia tecnología *blockchain*.

Esta tecnología, es muy reciente y avanza muy rápido, por lo que tienes que intentar estar siempre al día para ver los progresos o amenazas a las que se enfrenta.

Por otro lado, el último gran problema que he tenido que abordar ha sido la compilación del *smart contract*, desarrollado en Solidity, para poderlo transformar en un paquete para el lenguaje de programación *Go*. El protocolo de *Ethereum* cuenta con una implementación oficial en el lenguaje de *Go* [48] además en la librería estándar de este lenguaje podemos encontrar un paquete oficial de *Ethereum* con el que poder llevar a cabo muchas de las interacciones con la red de *Ethereum.*

El problema que tuve que sortear fue la creación del entorno de desarrollo para poder compilar los *contratos inteligentes* ya que, se ha de instalar el compilador junto a muchas herramientas y sus dependencias en forma de paquetes de *Go*, que utilizan dichas herramientas para funcionar.

Ya que el desarrollo del proyecto se ha realizado sobre el sistema operativo Windows, la primera aproximación fue descargar un contenedor de *Docker* para Windows, el cual se supone que incluye todo lo necesario para poder funcionar y comenzar a compilar contratos. Pero debido a que dicho contenedor no lo estaban manteniendo, daba muchos problemas de compatibilidad y no he conseguido hacerlo funcionar.

La siguiente opción fue también sobre el sistema operativo Windows, pero esta vez utilizando un paquete *npm* que podíamos encontrar para *Node.js*. En esta ocasión sí que conseguí instalar el compilador de *Solidity*, pero una vez instalado, para poder instalar las herramientas auxiliares que se requieren el proceso de compilación, hay que tener muchos programas que normalmente están destinados para Linux, instalados en Windows, como por ejemplo el compilador *GCC* y *CMake*. Todos estos programas pueden ser instalados en Windows, con cierta dificultad, el problema que se me presento en esta ocasión fue que *CMake* no funcionó una vez estaba instalado, por lo que no pude continuar adelante con esta aproximación.

Por ultimo y la aproximación que, sí ha funcionado, ha sido preparar todo el entorno de compilación en una máquina con sistema operativo Ubuntu, en el que se instaló *Go* y el compilador de *Solidity* junto con todas las herramientas necesarias. Aun así, tuve problemas con la dependencia de muchos de los paquetes que utilizaban las herramientas, y tuve que modificar dichos paquetes a mano para poder solucionar los problemas de

dependencias. Una vez resueltos todos los inconvenientes el compilador funcionó sin problemas y puede construir el paquete para el manejo de los *smart contracts* en *Go*.

6.2 Líneas futuras

En este apartado veremos las posibles líneas futuras con las que mejorar el proyecto ya que, el gran potencial de este proyecto nos permite un gran número de líneas futuras, que, si bien se quedan fuera del alcance de este proyecto, se describen a continuación. Podemos describir el proyecto actual, como el proyecto base sobre el que se debería construir más para tener un buen sistema de autenticación junto con la identidad de usuarios basadas en *blockchain*.

Además de las líneas futuras que veremos a continuación, también debemos de tener en cuenta que muchas de las líneas futuras del proyecto base a partir del cual se desarrolla este proyecto, siguen activas a fecha en la que se está escribiendo este documento.

6.2.1 Lista de clientes confiables

Durante el apartado *5.3.3.1 Servicio para interactuar con el smart contract* pudimos ver que la lista de los clientes en los cuales se confía está almacenada en la base de datos del cliente que ofrece el servicio. Para llevar a cabo esta tarea, se almacena en la base de datos del cliente registros con la dirección pública del cliente en el cual se quiere tener en cuenta, durante las comprobaciones pertinentes, así como un campo de tipo *bit* donde con un 0 indicaremos que no se confía y con un 1 indicaremos que se sí se confía.

De esta manera obtenemos una muy buena aproximación sobre los datos, ya que podemos realizar búsquedas rápidas en la base de datos, para comprobar por dirección, si se confía en un cliente externo o no.

La línea futura consistiría en desarrollar otro *smart contract* con el que poder almacenar y consultar la lista de clientes en los cuales confiamos. Esta aproximación y solución traería consigo todos los beneficios que aporta la tecnología de *blockchain*.

Para llevar a cabo esta aproximación, además de desarrollar un nuevo *smart contract* para que los usuarios pudieran utilizarlo de plantilla para sus propios contratos,

también se debería de desarrollar un servicio y funciones específicas para poder interactuar con dicho *smart contract* dentro del proyecto, de esta manera conseguiríamos un contrato en el que poder consultar dichos clientes confiables de modo que solo nosotros como cliente podamos escribir y consultar en el contrato inteligente.

6.2.2 Ficheros Keystores y HD Wallets

Como pudimos ver en el apartado *5.3.3.3 Creación de direcciones*, durante el proceso del registro de usuario el servicio permite la delegación de la creación de una dirección para un usuario, de modo parecido a una cuenta de usuario dentro de la *blockchain*. Pero en el desarrollo de este proyecto se optó por la primera aproximación vista, ya que ese era el primer método que se desarrolló, pero como vimos también en ese apartado a lo largo del tiempo, se habían desarrollado otras aproximaciones mejores a la originar para crear direcciones dentro de una *red blockchain*.

Una de esas aproximaciones era la del *fichero Keystore*, el cual es un fichero que cuenta con la clave privada, utilizada para la generación de la dirección pública, cifrada dentro del fichero por una contraseña. La seguridad del fichero era proporcional a la fuerza o seguridad de la contraseña. Este método puede llegar a ser más seguro si se utiliza una contraseña fuerte, ya que la primera aproximación desarrollada recordemos que enviaba la dirección tal cual al usuario.

En el apartado de generación de direcciones, también se menciona que disponemos de métodos para crear y cargar ficheros *keystore*. Pero como línea futura se han de desarrollar los mecanismos para poder enviar dicho fichero a través de la red y también se ha de crear la funcionalidad en la cual dicho fichero se elimine del servidor de manera correcta para que no queden restos. Además, el cliente deberá de cambiar la contraseña del fichero siempre y cuando sea posible.

Otra de las posibles aproximaciones son los *HD Wallets* o *monederos HD*. Estos tipos de monederos funcionan gracias a una semilla con la que se derivan todas las claves privadas que sean necesarias. De modo que como principal ventaja siempre que se disponga de la semilla, se podrá recuperar la clave privada, de ese modo no se perderían datos en caso de pérdida de la clave privada.

Se dispone por tanto como línea futura que, si el servicio no pudiera generar un fichero *keystore*, que por lo menos fuera capaz de generar las claves privadas mediante *HD Wallets*, de esta manera por lo menos se podrá recuperar las claves privadas emitidas por el servicio. Por ejemplo, esta situación se puede dar con personas que no son muy cercanas o afines a las nuevas tecnologías y las cuales pueden llegar a perder dichas claves privadas.

Las *HD Wallets* al ser un desarrollo posterior en el mundo de las *blockchain* y en concreto de *Ethereum* no está disponible en la implementación del protocolo de Ethereum para *Go*, pero sí que tenemos una manera de implementar los *HD Wallets* gracias un paquete llamado *go-ethereum-hdwallet* [49]. Este paquete implementa las interfaces las *wallets* disponibles en la implantación del protocolo en *Go*, por lo que son totalmente compatibles con el resto de los paquetes de este proyecto.

6.2.3 Roles, permisos y características

A lo largo de este documento se ha detallado que el *smart contract* cuenta con la capacidad de saber si un usuario ha sido autenticado, con qué fecha y por quién. Además de todo lo definido, como línea futura también se plantea introducir un sistema en cual los clientes puedan definir roles, permisos y características a tener en cuenta para el usuario.

Dentro de los roles, por ejemplo, un usuario podría ser un *visitante*, el cual solo interactúa con la página de manera que lee la información contenida en ella. También podríamos tener por ejemplo un usuario con el rol de *creador*, al cual se le permita escribir artículos en dentro de una web.

En el lado de los permisos, si seguimos con el mismo ejemplo de antes, podríamos contar con usuarios que son *visitantes* a los cuales a unos se les permite comentar dentro de la aplicación y a otros no. De nuevo siguiendo con este símil, se podría hacer que un usuario que es *creador* se le permita crear, pero no publicar información para un sistema de revisión, por ejemplo.

Por último, las características pueden ser muy diferentes entre clientes con propósitos dispares. Pero para esta línea futura se ha pensado en crear una lista de nombres de características, de manera que actúen como estándares en los diferentes permisos.

Para poder llevar a cabo esta línea futura se ha diseñado un sistema en el mediante programación en los *smart contracts* se podría añadir, actualizar y cambiar estos datos.

Esta información sería almacenada con el tipo *mapping*, muy similares a los mapas o diccionarios de otros idiomas, en *Solidity* y tratada en formato JSON, se han de realizar muchas pruebas para ver cuál es la mejor aproximación posible para esta línea futura.

6.2.4 Docker

Docker es un proyecto *OpenSource* que automatiza el despliegue de aplicaciones dentro de *contenedores* software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos. [50] [51].

Debemos de saber que *Docker* al igual que ocurre con sistemas de repositorio de código, cuenta con una plataforma para poder publicar imágenes de contenedores de manera pública y privada. Esto permite que se pueda tener un repositorio con una imagen de este servicio de manera pública para todo el mundo.

Como el servicio que se ha desarrollado en este proyecto puede llegar a ser un poco complejo a la hora de implantarlo en nuestros sistemas, además, de que ha sido desarrollado en lenguajes modernos y cuanta con mucha dependencia de paquetes propios del lenguaje y su librería estándar, se ha pensado en montar todo el servicio en un *contenedor Docker* de manera que, manteniendo una imagen del servicio, este pueda ser utilizada en cualquier sistema.

Esta solución, también permitiría a los clientes levantar el servicio en sus sistemas con muy pocos comandos o configuración. Además, permitiría la importación rápida a sistemas *cloud* como *Azure* o *AWS*.

6.2.5 Verificación de la firma ECDSA mediante un *smart contract.*

Como hemos podido comprobar en el apartado de autenticación mediante *single sign on*, la comprobación de la firma la realiza el cliente. Pero debido al coste computacional, la rapidez con la que se pueda comprobar la firma y la seguridad se ha propuesto que esta comprobación de la firma la realice el propio *smart contract*.

Para poder llevar a cabo esta funcionalidad en *Solidity*, deberemos de usar de nuevo la función *ecrecover* con la cual podremos recuperar la dirección pública. En la función de *Solidity* deberemos de generar una función privada, con el decorador de función *pure*, la cual devuelva la dirección dándole el la firma y el desafío firmado. En dicha función se realizará una disección de la firma en los componentes {r,s} y junto a estos se recuperará la dirección.

De esta manera se consigue que toda la verificación de la firma, así como la comprobación de la dirección del usuario, se lleve a cabo en el *smart contract*. Además, con esta línea futura, también se pretende que el resto de las comprobaciones también la realice el *smart contract*, como por ejemplo la comprobación de la fecha de expiración. De este modo el cliente que desee verificar a un usuario para la autenticación *single sign on* solo deberá de consultar el contrato del usuario.

7. Referencias

[1] «Identity Management Wikipedia,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Identity_management. [Último acceso: 10 2 2020].

[2] L. P. Sevilla, «Servicio de autentificaicon de usuarios,» 2019. [En línea]. Disponible: https://rua.ua.es/dspace/handle/10045/93590.

[3] «Ethereum Wikipedia,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Ethereum. [Último acceso: 20 4 2020].

[4] M. Mukhopadhyay, «Grokking Ethereum,» de *Ethereum Smart Contract Development*, Packt Publishing, 2018.

[5] «Blockchain Wikipedia,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Blockchain. [Último acceso: 20 4 2020].

[6] M. Mukhopadhyay, «Blockchain Basics,» de *Ethereum Smart Contract Development*, Packt Publishing, 2018.

[7] «Sovrin,» [En línea]. Disponible: https://sovrin.org/blog/. [Último acceso: 10 2 2020].

[8] «Hyperleger,» [En línea]. Disponible: https://www.hyperledger.org/. [Último acceso: 16 2 2020].

[9] «bytecode,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Bytecode. [Último acceso: 10 5 2020].

[10] «Smart Contracts Wikipedia,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Smart_contract. [Último acceso: 20 2 2020].

[11] M. Mukhopadhyay, «Deep-Diving into Smart Contracts,» de *Ethereum Smart Contract Development*, Packt Publishing, 2018.

[12] «SingleSignOn,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Single_sign-on. [Último acceso: 10 3 2020].

[13] «JWT,» [En línea]. Disponible: https://en.wikipedia.org/wiki/JSON_Web_Token. [Último acceso: 10 4 2020].

[14] «Argon2,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Argon2. [Último acceso: 2 5 2020].

[15] «PasswordHashingCompetition,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Password_Hashing_Competition. [Último acceso: 25 6 2020].

[16] «Argon2 Documentation,» [En línea]. Disponible: https://www.cryptolux.org/images/0/0d/Argon2.pdf. [Último acceso: 2 5 2020].

[17] «bcrypt,» [En línea]. Disponible: https://www.usenix.org/legacy/event/usenix99/provos/provos.pdf. [Último acceso: 10 5 2020].

[18] «PBKDF2,» [En línea]. Disponible: https://tools.ietf.org/html/rfc2898. [Último acceso: 5 10 2020].

[19] «Metodología Desarrollo Software,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Software_development_process. [Último acceso: 12 2 2020].

[20] «REST Wikipedia,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Representational_state_transfer. [Último acceso: 10 3 2020].

[21] «Go,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Go_(programming_language). [Último acceso: 10 3 2020].

[22] N. Yellavula, Building RESTful Web services with Go, Packt Publishing, 2017.

[23] «VSC,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Visual_Studio_Code. [Último acceso: 2 5 2020].

[24] «Remix Ethereum IDE,» [En línea]. Disponible: http://remix.ethereum.org. [Último acceso: 14 2 2020].

[25] «Ganache,» [En línea]. Disponible: https://github.com/trufflesuite/ganache. [Último acceso: 10 3 2020].

[26] «SQL Server,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Microsoft_SQL_Server. [Último acceso: 12 4 2020].

[27] «go-mssqldb,» [En línea]. Disponible: https://github.com/denisenkom/go-mssqldb. [Último

acceso: 10 3 2020].

[28] «GPL,» [En línea]. Disponible: https://en.wikipedia.org/wiki/GNU_General_Public_License. [Último acceso: 10 3 2020].

[29] L. P. Sevilla, «GitHub,» 7 9 2020. [En línea]. Disponible: https://github.com/Lixto/IdentityManagementGoAPI. [Último acceso: 9 7 2020].

[30] S. 5.3. [En línea]. Disponible: https://solidity.readthedocs.io/en/v0.5.3/layout-of-source-files.html. [Último acceso: 2 5 2020].

[31] M. Mukhopadhyay, «Solidity in Depth,» de *Ethereum Smart Contract Development*, Packt Publishing, 2018.

[32] «Modifier en Solidity,» [En línea]. Disponible: https://solidity.readthedocs.io/en/v0.6.8/structure-of-a-contract.html?highlight=modifier#function-modifiers. [Último acceso: 10 3 2020].

[33] «Unix time,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Unix_time. [Último acceso: 10 4 2020].

[34] «Snap,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Snap_(package_manager). [Último acceso: 20 5 2020].

[35] «Sal críptografica,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Salt_(cryptography). [Último acceso: 25 6 2020].

[36] «Argon2,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Argon2.

[37] SHA3. [En línea]. Disponible: https://en.wikipedia.org/wiki/SHA-3. [Último acceso: 16 4 2020].

[38] «HD Wallets,» [En línea]. Disponible: https://academy.bit2me.com/que-son-las-hd-wallets/. [Último acceso: 10 4 2020].

[39] «Bip32,» [En línea]. Disponible: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki. [Último acceso: 1 6 2020].

[40] «jwtGo,» [En línea]. Disponible: https://github.com/dgrijalva/jwt-go. [Último acceso: 10 4 2020].

[41] «ECDSA,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm. [Último acceso: 20 4 2020].

[42] K. Raj, Foundations of Blockchain, Packt Publishing, 2019.

[43] «ECDSA Sing,» [En línea]. Disponible: https://tools.ietf.org/html/rfc6979#section-3.2. [Último acceso: 20 4 2020].

[44] «ecrecover,» [En línea]. Disponible: https://solidity.readthedocs.io/en/latest/units-and-global-variables.html#mathematical-and-cryptographic-functions. [Último acceso: 30 4 2020].

[45] «cryptox509,» [En línea]. Disponible: https://golang.org/pkg/crypto/x509/. [Último acceso: 10 5 2020].

[46] «ITU,» [En línea]. Disponible: https://www.itu.int/es/Pages/default.aspx. [Último acceso: 20 4 2020].

[47] «Ethereum,» [En línea]. Disponible: https://ethereum.org/. [Último acceso: 20 4 2020].

[48] «GoEthereum,» 10 3 2020. [En línea]. Disponible: https://github.com/ethereum/go-ethereum.

[49] «HDWalletsGo,» [En línea]. Disponible: https://github.com/miguelmota/go-ethereum-hdwallet. [Último acceso: 11 4 2020].

[50] «Docker,» [En línea]. Disponible: https://en.wikipedia.org/wiki/Docker_(software). [Último acceso: 1 6 2020].

[51] «Docker2,» [En línea]. Disponible: https://www.docker.com/why-docker. [Último acceso: 1 6 2020].

[52] «Password Hashing Competition,» [En línea]. Disponible: https://password-hashing.net/#argon2. [Último acceso: 25 6 2020].

8. Anexo

En este anexo se muestra la documentación para poder hacer uso del servicio.

Esta documentación, va a ser añadida en formato *JSON*, para poder interpretarlo mejor ya que, de esta manera podremos ver como se han de formar el cuerpo y las cabeceras de las peticiones. Además, gracias a este formato pueden se importadas a programas como *Postman* con el que se pueden realizar pruebas.

```
"name": "User Register",
"request": {
    "method": "POST",
    "header": [
        {
            "key": "Content-Type",
            "value": "application/json",
            "type": "text"
        }
    ],
    "body": {
        "mode": "raw",
        "raw": "{\n\t\"Email\": \"demo\",\n\t\"Password\" : \"demo\",\n t\"Address\" : \"0xf17a028726327195Df0a2B0c76a50475e53D9bCd\"\n}",
        "options": {
            "raw": {
                "language": "json"
            }
        }
    },
    "url": {
        "raw": "{{URL}}/register",
        "host": [
            "{{URL}}"
        ],
        "path": [
            "register"
        ]
    }
}
```


Llamada 1 Registro de usuarios

```
"name": "User Login",
"request": {
    "method": "POST",
    "header": [
        {
            "key": "Content-Type",
            "value": "application/json",
            "type": "text"
        }
    ],
    "body": {
        "mode": "raw",
        "raw": "{\"Email\": \"demo\",\"Password\" : \"demo\",
            \"Address\":\"0xb161eef96b1FbDc980a5363f670ab460fb04548d\",
            \"Contract\": \"0xb5175ef349d45fB71E63394b19e75C\",
            \"PublicKey\": \"0xb5175ef349d45fB71E63394b19e75C\"} }",
        "options": {
            "raw": {
                "language": "json"
            }
        }
    },
    "url": {
        "raw": "{{URL}}/login",
        "host": [
            "{{URL}}"
        ],
        "path": [
            "login"
        ]
    }
}
```

Llamada 2 Autenticación de usuarios

```
"name": "InsertMultiFactor",
"request": {
    "method": "POST",
    "header": [
        {
            "key": "Content-Type",
            "value": "application/json",
            "type": "text"
        }
    ],
    "body": {
        "mode": "raw",
        "raw": "{\n\t\"Email\": \"demo\",\n\t\"Password\" : \"demo\",\n\t\"Fingerprint\" : true\n}",
        "options": {
            "raw": {
                "language": "json"
            }
        }
    },
    "url": {
        "raw": "{{URL}}/insertMultiFactor",
        "host": [
            "{{URL}}"
        ],
        "path": [
            "insertMultiFactor"
        ]
    }
},
```


Llamada 3 Inserción de valor para un segundo factor de autenticación

```
"name": "Check Token",
"protocolProfileBehavior": {
    "disableBodyPruning": true
},
"request": {
    "method": "GET",
    "header": [
        {
            "key": "Authorization",
            "value": "{{token}}",
            "type": "text"
        },
        {
            "key": "Content-Type",
            "value": "application/json",
            "type": "text"
        }
    ],
    "body": {
        "mode": "raw",
        "raw": ""
    },
    "url": {
        "raw": "{{URL}}/checkToken",
        "host": [
            "{{URL}}"
        ],
        "path": [
            "checkToken"
        ]
    }
}
```


Llamada 4 Comprobación del token

```
"name": "User Login Sign On",
"request": {
    "method": "POST",
    "header": [
        {
            "key": "Content-Type",
            "value": "application/json",
            "type": "text"
        },
        {
            "key": "Authorization",
            "value": "{{token}}",
            "type": "text"
        }
    ],
    "body": {
        "mode": "raw",
        "raw": "{\"Contract\":\"0xb5175ef349d45fB71E63394b19e75C\",\"Signature\": \"0xb5175ef349d45fB71E63394b19e75C\"} }",
        "options": {
            "raw": {
                "language": "json"
            }
        }
    },
    "url": {
        "raw": "{{URL}}/login",
        "host": [
            "{{URL}}"
        ],
        "path": [
            "login"
        ]
    }
}
```


Llamada 5 Autenticación single sign on

```
"name": "User Logout ",
"request": {
    "method": "POST",
    "header": [
        {
            "key": "Content-Type",
            "value": "application/json",
            "type": "text"
        },
        {
            "key": "Authorization",
            "value": "{{token}}",
            "type": "text"
        }
    ],
    "body": {
        "mode": "raw",
        "raw": "{\n\t\"Email\": \"demo\",\n\t\"Password\" : \"demo\",\n\t\"Fingerprint\" : false\n}"
    },
    "url": {
        "raw": "{{URL}}/logout",
        "host": [
            "{{URL}}"
        ],
        "path": [
            "logout"
        ]
    }
}
```


Llamada 6 Cierre de sesión

```
"name": "Get Trust Users",
"protocolProfileBehavior": {
    "disableBodyPruning": true
},
"request": {
    "method": "GET",
    "header": [
        {
            "key": "Authorization",
            "value": "{{token}}",
            "type": "text"
        }
    ],
    "url": {
        "raw": "{{URL}}/users",
        "host": [
            "{{URL}}"
        ],
        "path": [
            "users"
        ]
    }
}
```


Llamada 7 Recuperación de la lista de clientes confiables